МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Долголенко О. М.

ПРОЄКТУВАННЯ СКЛАДНИХ СИСТЕМ

Навчальний посібник

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського як навчальний посібник для здобувачів ступеня бакалавра за освітньою програмою «Інженерія програмного забезпечення комп'ютерних систем» спеціальності 121 Інженерія програмного забезпечення

Електронне мережеве навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2023

УДК 004.3

Автор *Долголенко Одександо Миколайович,* канд. техн. наук, с.н.с.

Рецензент *Катін П. Ю.,* канд. техн. наук, доц., доцент кафедри радіотехнічних систем РТФ , Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського»

Відповідальний редактор *Луцький Г.М.,* д.т.н., проф. кафедри обчислювальної техніки ФІОТ

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 3 від 07.12.2023 р.)
за поданням Вченої ради факультету інформатики та обчислювальної техніки
(протокол № 3 від 30.10.2023 р.)

Долголенко О. М.

Проєктування складних систем [Електронний ресурс]: навч. посіб. для студ. спеціальності 121 «Інженерія програмного забезпечення» / КПІ ім. Ігоря Сікорського; автор: О.М. Долголенко. – Електронні текстові дані (1 файл: 2,84 Мбайт). – Київ: КПІ ім. Ігоря Сікорського, 2023. – 102 с.

В навчальному посібнику розглянуті питання проєктування складних систем на яких, скоріше за все, будуть працювати паралельні програмісти при здійсненні своєї професійної діяльності – це сучасні мікропроцесорні ядра з суперскалярною архітектурою.

Для студентів всіх форм навчання, які навчаються у бакалавраті за спеціальністю 121 «Інженерія програмного забезпечення» факультету інформатики та обчислювальної техніки, а також інших споріднених спеціальностей.

Може бути корисним для дипломного проектування, оскільки містить технічну інформацію з побудови та функціонування вузлів суперскалярних мікропроцесорних ядра та особливостей функціонування вбудованих інноваційних технологій.

УДК 004.3

Реєстр. № НП 23/24-166. Обсяг 3,6 авт. арк.
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Берестейський, 37, м. Київ, 03056 https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

ЗМІСТ

ПЕРЕДМОВА

Навчальний посібник призначений для того, щоб навчити студентів розуміти яким чином їхня паралельна програма буде виконуватися в сучасному мікропроцесорному ядрі із суперскалярною архітектурою.

Сучасні мікропроцесорні ядра, після того як стали суперскалярними, стали настільки складними, що лише незначна кількість спеціалістів, крім, наприклад, співробітників науково-дослідного центру фірми Intel, що знаходиться в місті Хайфа (Ізраїль) і займається розробкою нових мікроархітектур, розуміють, в деякому наближенні, подробиці їхньої роботи. Так, більшість університетів світу вивчають особливості роботи сучасних мікропроцесорних ядер по книзі John L. Hennessy, David A. Patterson. Computer Architecture: A Cuantitative Approach 6th Edition, USA, Morgan Kaufmann, 2019-1527 p., яка витримала вже шосту редакцію і має промовисту назву A Cuantitative Approach.

При написанні цього навчального посібника використані результати науково-дослідної роботи «Розробка теоретичних основ побудови високопродуктивних комп’ютерних систем з динамічним розпаралелюванням обчислювальних процесів», Закл. звіт по НДР № ДР 0213U004830, -Київ, 2013, виконаної на кафедрі Обчислювальної техніки КПІ ім. Ігоря Сікорського під керівництвом проф. Георгія Михайловича Луцького. Автор висловлює глибоку подяку колишнім науковим співробітникам кафедри: Аксьоненко Сергію Володимировичу та Бліновій Тетяні Олександрівні за їхній вклад в розуміння особливостей функціонування суперскалярного мікропроцесорного ядра.

1 АРХІТЕКТУРНІ ОСОБЛИВОСТІ ПОБУДОВИ ЯДРА СУЧАСНОГО СУПЕРСКАЛЯРНОГО МІКРОПРОЦЕСОРА

1.1 Види паралелізму в роботі ядра сучасних мікропроцесорів

В роботі ядра сучасних мікропроцесорів, з точки зору програміста, можливо виділити наступні види паралелізму [1]:

DLP (Data-Level Parallelism або ***SIMD****, <SIMD>::=<Single Instruction Multiply Dates>)* – паралелізм на рівні даних*;*

ILP (Instructions-Level Parallelism) – паралелізм на рівні команд*;*

Threads-Level Parallelism (у *Intel* це *Hyper-Threading*) – паралелізм на рівні потоків команд*;*

Fibers-Level Parallelism – паралелізм на рівні файберів (потік команд може складатися з обробки деякої множини файберів).

Замість 2-ух останніх термінів, в паралельному програмуванні використовується термін ***Tasks-Level Parallelism (TLP)*** [2], де *<Task>::=<Thread>|<Fiber>*.

При описуванні архітектури *x86-64* [2], *Intel* використовує поняття *Task*, як *<Task>::=<a unit of work that a processor can dispatch, execute, and suspend>*.

1.2 Паралелізм на рівні даних в архітектурі *x86-64*

<SIMD в архітектурі *x86-64>::=<*$MMX_{64/8}$*>|<*$3DNow!_{64/8}$*>|<Enhanced* $3DNow!_{64/8}$*>)|<*$SSE_{128/8}$*>|<*$SSE2_{128/8}$*>|<*$SSE3_{128/8}$*>|<*$SSSE3_{128/16}$*>|<*$SSE4_{128/16}$*>(<* $SSE4.1_{128/16}$*>+<*$SSE4.2_{128/16}$*>)|<*$AVX_{256/16}$*>|<*$AVX2_{256/16}$*>|<*$FMA4_{256/16}$*>*(з AMD Bulldozer, 2011)|<*$FMA3_{256/16}$*>*(з: AMD Piledriver, 2012; Intel Haswell, 2013) |<*$AVX512_{512/32}$*>.

MMX *(Multimedia Extensions)*. Набір інструкцій, що був розроблений для прискорення процесів кодування/декодування потокових аудіо та відеоданих. Він є розвитком технології, запропонованої в мікропроцесорі *i860*. Розроблений у лабораторії ***Intel*** в Хайфі, Ізраїль, в першій половині 1990-х. Впроваджений в *Pentium MMX (1997, Intel P5).*

Розширення MMX включають в себе вісім 64-бітних регістрів загального використання MM0—MM7. Для сумісності із способами переключення контексту процесора в існуючих ОС *Intel* була вимушена поєднати в програмній моделі процесора вісім регістрів *MMX* з вісьмома регістрами *FPU* (математичного співпроцесора). Апаратно це можуть бути різні пристрої, але з точки зору програміста — це одні і ті ж регістри. Таким чином, було неможливим одночасне використання команд математичного співпроцесору та команд *MMX*.

MMX містить 54 інструкції для виконання **цілочислових** операцій **одинарної точності**. Цей набір інструкцій не отримав широкого розповсюдження, хоча його підтримка вбудована у всі сучасні процесори.

3DNow! - в 1998 р. *AMD* презентувала новий процесор *K6-2*, котрий мав мультимедійний набір команд *3DNow!*, що має 21 інструкцію для виконання операцій **з плаваючою крапкою одинарної точності**. Презентація технології відбулася одночасно з виходом гри *Quake2*, де ця технологія була вперше використана. Приріст продуктивності при її включенні складав до 30%. Це був прорив: *AMD* вперше (але не в останній раз) випередила *Intel* в представленні нового індустріального стандарту. *3DNow!* підтримали багаточисельні виробники апаратного і програмного забезпечення (в тому числі *Microsoft*, всі основні виробники відеокарт: *NVidia, 3Dfx, S3, ATI*).

Enhanced 3DNow! – вперше впроваджені в сімействі *AMD, K7(Athlon, 1999 р.).* Набор містить 19 мультимедійних інструкцій (перетворення упакованих форматів операндів, їх обробки та збереження вмісту *MMX* регістрів в пам'яті) і 5 *DSP (Digital Signal Processor)* інструкцій. Ці мультимедійні інструкції підняли розуміння *3D* графіки на новий рівень. 5 *DSP* інструкцій допомогли знизить загрузку процесора при використанні софт-модема, програвання *MP3* файлів, софт-*ADSL*, звука *Dolby Digital.*

SSE інструкції – впроваджені *Intel* в сімействі *Pentium 3 (*ядро *Katmai*, кінець 1999 р.). *SSE* містить 71 мультимедійну інструкцію (в тому числі всі інструкції з наборів: *3DNow!* та *Enhanced 3DNow!*, за винятком 5 *DSP* інструкцій). Крім нових мультимедійних інструкцій, в кожний *Pentium 3* був вбудований

ідентифікаційний номер, котрий дозволяв відслідковувати користувача. В одній із країн Європейського Союзу, із-за цього номеру, була введена заборона на ввезення *Pentium 3*. В цілому, наявність цього номера уповільнило впровадження *SSE* і часто сприяло вибору процесора на користь *Athlon*.

Технологія *SSE* дозволила перебороти дві основні проблеми *MMX*: при використанні *MMX* неможливо було одночасно використовувати інструкції співпроцесора, так як його регістри, програмно, були загальними з регістрами *MMX*, та можливість *MMX* працювати тільки з цілими числами.

Технологія *SSE* **включила в архітектуру *x86* вісім 128-бітних регістрів (*XMM(SSE)0-XMM(SSE)7*) та набір інструкцій, що можуть працювати із скалярними і векторними даними з плаваючою крапкою одинарної точності, а також упакованими типами даних. Вісім 64-бітних регістрів *MM0—MM7* з цього моменту стали молодшими половинками, відповідно, 128-бітних регістрів (*XMM(SSE)0-XMM(SSE)7*).**

Перевага в продуктивності досягається в тому випадку, коли необхідно виконати одну й ту же послідовність дій над різними даними. В такому випадку блоком *SSE* здійснюється розгалуження обчислювального процесу на рівні даних.

SSE2 – набор із 144 інструкцій, розроблений *Intel* і вперше впроваджений в процесорах серії *Pentium 4* (ядро *NetBurst,* 2000 р.). *SSE2* розширює набір інструкцій *SSE* з метою повністю витіснити *MMX*. *SSE2* включає в себе набір інструкцій, котрий виконує операції із скалярними з плаваючою крапкою, векторними та упакованими типами даних.

- *SSE2* містить інструкції для потокової обробки цілочислових даних в тих же 128-бітних *XMM(SSE)* регістрах, що робить це розширення більш кращим для цілочислових обчислень, ніж використання набора інструкцій *MMX*.
- *SSE2* працює з дійсними числами.
- *SSE2* включає в себе ряд команд керування кешем, призначених для мінімізації засмічування кеша при обробці об’ємних потоків даних.
- *SSE2* включає в себе складні доповнення до команд перетворення чисел.

- *FPU (x87)* інструкції забезпечують вищу точність завдяки обчисленню проміжних результатів із точністю 80 біт, за замовчуванням, щоб мінімізувати помилку округлення в чисельно нестабільних алгоритмах (див. обґрунтування стандарту IEEE 754 і посилання в ньому [3]). Однак *FPU x87* є лише скалярним пристроєм, тоді як *SSE2* може обробляти невеликий вектор операндів паралельно.

- Якщо коди, розроблені для *x87*, переносяться на *SSE2* із плаваючою точкою нижчої точності, певні комбінації математичних операцій або вхідних наборів даних можуть призвести до числових відхилень у менш значущих розрядах мантіси результату, що може бути проблемою у відтворюванні наукових обчислень, напр. якщо результати обчислень необхідно порівнювати з результатами, отриманими на обчислювальній машині з іншої архітектурою. Проблема полягає в тому, що історично стандарти мови та компілятори були непослідовними у своїй обробці 80-розрядних регістрів *x87*, що реалізовували змінні подвійної розширеної точності, порівняно з подвійними проміжними значеннями, що могли зберігатися в пам'яті.

- *SSE*2 використовує вісім 128-бітних регістрів (*XMM(SSE)0 - XMM(SSE)7*), кажен із котрих трактується як:

- 128-бітний упакований тип даних з плаваючою крапкою подвійної точності;

- 128-бітний упакований тип даних з плаваючою крапкою одинарної точності;

- два 64-бітних значення з плаваючою крапкою подвійної точності;
- чотири 32-бітних значення з плаваючою крапкою одинарної точності;
- два 64-бітних значення з фіксованою крапкою типу *long*;
- чотири 32-бітних значення з фіксованою крапкою типу *int*;
- вісім 16-бітних значень з фіксованою крапкою типу *short;*
- шістнадцять 8-бітних значень з фіксованою крапкою типу *byte*;
- скаляр з плаваючою крапкою;
- 32-бітний регістр прапорів (*MXCSR*).

SSE3 – набір із 13 інструкцій, розроблений *Intel* і представлений в процесорах серії *Pentium 4* (ядро *Prescot,* 2004 р.). Найбільш помітні зміни — ***можливість горизонтальної робото з регістрами.*** Додані команди додавання і віднімання декількох значень, що зберігаються в одному регістрі (Векторні операції, результатом виконання котрих є скаляр). Ці команди спростили ряд *DSP*- та *3D*-операцій. Існує також нова команда для перетворення значень з плаваючою крапкою в цілі без необхідності вносити зміни в глобальному режимі округлення.

SSSE3 або ***SSE3S*** – набір із 16 інструкцій, розроблених *Intel* і впроваджений в процесорах з мікроархітектурою *Core* (ядро "*Woodcrest" Xeons,* 2006 р., архітектура *x86-64*). ***Кожна із команд може працювати як з 64-х бітними (MMX), так і з 128-ми бітними (XMM) регістрами***, тому *Intel* в своїх матеріалах посилається на 32 нові команди:

- Дванадцять інструкцій, які виконують горизонтальні операції додавання або віднімання.
- Шість інструкцій, які оцінюють абсолютні значення.
- Дві інструкції, які виконують операції множення та додавання та прискорюють обчислення скалярних добутків.
- Дві інструкції, які прискорюють операції множення упакованих цілих чисел та створюють цілі значення результату з масштабуванням.
- Дві інструкції, які виконують побайтове перетасування операнду на місці відповідно до другого операнду, що керує перетасуванням.
- Шість інструкцій, які заперечують упаковані цілі числа в операнді призначення, якщо знак відповідного елемента в операнді джерела менше нуля.
- Дві інструкції, які вирівнюють дані з двох операндів.

SSE4 – складається з 54 інструкцій. Підмножина, що складається з 47 інструкцій, яка в деякій документації Intel називається *SSE4.1*, стала доступною з *Penryn* (2007). Крім того, *SSE4.2*, другий піднабір, що складається з 7 інших інструкцій, вперше доступний у *Nehalem* (2008).

Починаючи з процесорів на основі *Barcelona* (2007), AMD представила набір інструкцій *SSE4a*, який містить 4 інструкції *SSE4* і 4 нові інструкції *SSE*. Цих інструкцій немає в процесорах Intel. Процесори AMD почали підтримувати *SSE4.1* і *SSE4.2* Intel (повний набір інструкцій *SSE4*) лише в процесорах *FX* на базі *Bulldozer* (2011). З *SSE4a* також була представлена функція невирівняного *SSE*, що означало, що невирівняні інструкції завантаження були такими ж швидкими, як і вирівняні версії на вирівняних адресах. Це також дозволило вимкнути перевірку вирівнювання для операцій *SSE* без завантаження, які звертаються до пам'яті. Пізніше Intel запровадив подібні покращення швидкості для невирівняного *SSE* у своїх процесорах *Nehalem*, але не запровадив невирівняний доступ за допомогою інструкцій *SSE* без завантаження до *AVX*.

В *SSE4* додані інструкції, що пришвидшують компенсацію руху у відеокодеках, швидке читання з *USWC* пам'яті (*Uncacheable speculative write combining, is a computer BIOS setting for memory communication between CPU and graphics card*), множина інструкцій для спрощення векторизації програм компіляторами. Крім того, в *SSE4.2* додані інструкції обробки строк 8/16-бітних символів, обчислення *CRC32*, *POPCNT* (повертає кількість бітів, установлених в 1). Вперше в *SSE4* регістр *XMM0* став використовуватися як неявний аргумент для деяких інструкцій.

AVX (*Advanced Vector Extensions)* – Набір інструкцій *Intel* (з *Sandy Bridge*, 2011) розширює 128-розрядні набори інструкцій *SIMD*, використовуючи нову схему кодування інструкцій через векторний префікс розширення (*VEX*). Intel *AVX* також пропонує кілька вдосконалених функцій, окрім тих, які були доступні в попередніх поколіннях 128-розрядних розширень *SIMD*. Ширину файлу регістрів *SIMD* збільшено зі 128 біт до 256 біт і перейменовано з *XMM0–XMM7* на *YMM0–YMM7* (у режимі *x86-64*: *YMM0–YMM15*). У процесорах із підтримкою *AVX* застарілі інструкції *SSE* (які раніше працювали на 128-розрядних регістрах *XMM*) можна розширити за допомогою префіксу *VEX* для роботи з молодшими 128 бітами регістрів *YMM*.

Intel *AVX* представляє наступні вдосконалення архітектури:

• Підтримка 256-бітних векторів із набором векторних регістрів *YMM*.

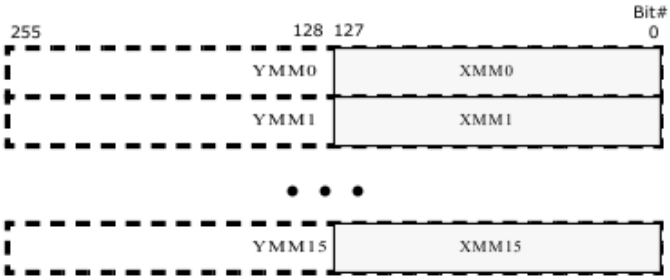

Рисунок 1.1 – 256-бітний *SIMD* регістр

• Покращення 256-бітного набору інструкцій з плаваючою крапкою з підвищенням продуктивності до 2 разів порівняно зі 128-бітними розширеннями *SIMD*.

• Удосконалення застарілих 128-розрядних розширень інструкцій *SIMD* для підтримки синтаксису трьох операндів і спрощення векторизації компілятором виразів мови високого рівня. Тепер компілятори замість $a=a+b$ можуть використовувати $c=a+b$, при цьому регістр a остається незмінним. У випадках, коли значення a використовується в наступних обчисленнях, це підвищує продуктивність, тому що позбавляє від необхідності зберігати перед обчисленням та поновлювати після обчислень регістр, що містить a, із іншого регістра чи пам’яті.

• Підтримка синтаксису інструкцій із префіксним кодуванням *VEX* для узагальненого синтаксису трьох операндів для покращення гнучкості програмування інструкцій та ефективного кодування нових розширень інструкцій.

• Більшість *VEX*-кодованих 128-бітних і 256-бітних інструкцій *AVX* (як із семантикою завантаження, так і з семантикою обчислювальної операції) не обмежені 16-байтним або 32-байтовим вирівнюванням пам’яті. Для створення нових інструкцій відсутня вимога до необхідності вирівнювання операндів у пам’яті. Однак зазвичай слідкувати за вирівнюванням на розмір операндів, для запобігання значного зниження продуктивності.

• Підтримка гнучкого розгортання: 256-бітного коду *AVX*, 128-бітного коду *AVX*, застарілого 128-бітного коду та скалярного коду.

За винятком інструкцій *SIMD*, що працюють на регістрах *MMX*, майже всі застарілі 128-розрядні інструкції *SIMD* мають еквіваленти *AVX*, які підтримують синтаксис трьох операндів. 256-розрядні інструкції *AVX* використовують синтаксис із трьома операндами, а деякі з синтаксисом із 4 операндами. Кожен регістр *YMM* може містити:

• вісім 32-розрядних чисел одинарної точності з плаваючою комою;

• чотири 64-розрядних числа з плаваючою комою подвійної точності;

• скаляр з плаваючою крапкою.

FMA (*Fused-Multiply-ADD)* – набір інструкцій є розширенням 128- і 256-бітових *SIMD* інструкцій. Є два варіанти цих інструкцій:

- ***FMA4*** підтримується процесорами *AMD*, починаючи з мікроархітектури *Bulldozer (2011). FMA4* був реалізованим в «залізі» раніше ніж *FMA3*;
- ***FMA3*** підтримується процесорами *AMD*, починаючи з мікроархітектури *Piledriver (2012)* і процесорами *Intel*, починаючи з мікроархітектури *Haswell (2013)*.

FMA3 і *FMA4* команди мають майже ідентичні функції, але не сумісні. Обидві містять інструкції *FMA* для скалярних операцій з плаваючою крапкою та операцій *SIMD*, але інструкції *FMA3* (складаються з 20 інструкцій) мають три операнди, тоді як інструкції *FMA4* (складаються з 6 інструкцій) мають чотири операнди. Операція *FMA* має вигляд $\boldsymbol{d = round(a \cdot b + c)}$, де функція ***round*** виконує округлення, щоб дозволити результату поміститися в регістр призначення, якщо є забагато значущих бітів, щоб поміститися в адресат.

Форма з чотирма операндами (*FMA4*) дозволяє *a, b, c* і *d* бути чотирма різними регістрами, тоді як форма з трьома операндами (*FMA3*) вимагає, щоб *d* був тим самим регістром, що й *a, b* або *c*. Форма з трьома операндами робить код коротшим, а апаратну реалізацію трохи простішою, тоді як форма з чотирма операндами забезпечує більшу гнучкість програмування.

AVX2 – набір інструкцій від Intel (з "*Haswell", 2013*), також відомий як нові інструкції *Haswell* (містить 15 команд), є розширенням набору інструкцій *AVX*, впровадженим в мікроархітектурі *Haswell*. Набір *AVX2* робить такі доповнення:

• розширює більшість векторних цілочисельних інструкцій *SSE* та *AVX* до 256 біт;

• здійснює бітові маніпуляції та множення загального призначення з трьома операндами;

• підтримує тип адресації пам’яті *Gather*, котрий часто виникає при адресації векторів в розріджених операціях лінійної алгебри і дозволяє швидко завантажувати векторні елементи з несуміжних ділянок пам'яті;

- *DWORD* (4 байти) і *QWORD* (8 байтів) - зернистість будь-яких (*any-to-any*) перестановок;
- векторні зсуви.

AVX512 – набір інструкцій від Intel (з "*Knights Landing*"-*Xeon Phi, 2016). AVX-512* інструкції закодовані за допомогою нового префіксу *EVEX*. Цей набір допускає: 4 операнди, 7 нових 64-розрядних регістрів маски операнду, режим скалярної пам’яті з автоматичною трансляцією, явне керування округленням і режим адресації пам’яті зі стислим зміщенням. **Ширина регістрового файлу збільшена до 512 біт, а загальна кількість регістрів збільшена до 32 (регістри *ZMM0-ZMM31*) у режимі *x86-64***. *AVX-512* складається з кількох розширень, не всі призначені для підтримки всіма процесорами, які їх реалізують. Набір інструкцій складається з наступних розширень:

- *AVX-512 основні (F)* – додає кілька нових інструкцій і розширює більшість 32-бітних і 64-бітних інструкцій з плаваючою крапкою *SSE-SSE4.1* і *AVX/AVX2* схемою кодування *EVEX* для підтримки 512-бітних регістрів, масок операцій, трансляції параметрів, а також вбудованого округлення та контролю винятків;
- *AVX-512 інструкції виявлення конфліктів (CD)* – ефективне виявлення конфліктів, що дозволяє векторизувати більше петель, підтримується *Knights Landing*;

- *AVX-512 експоненціальні та взаємні операції (ER)* – призначені для допомоги в реалізації трансцендентних операцій, підтримується *Knights Landing*;
- *AVX-512 інструкції попередньої вибірки (PF)* – нові можливості попередньої вибірки, підтримується *Knights Landing*;
- *AVX-512 розширення довжини вектора (VL)* – розширює більшість операцій *AVX-512* для роботи з регістрами *XMM* (128-біт) і *YMM* (256-біт) (включаючи *XMM16-XMM31* і *YMM16-YMM31* у режимі *x86-64*);
- *AVX-512 байтові та словесні інструкції (BW)* – розширює *AVX-512* для охоплення 8-бітних і 16-бітних цілочисельних операцій;
- *AVX-512 Doubleword і Quadword інструкції (DQ)* – покращені 32-бітні та 64-бітні операції з цілими числами;
- *AVX-512 Integer Fused Multiply Add (IFMA)* – об'єднане додавання множення для 512-бітних цілих чисел;
- *AVX-512 Vector Byte Manipulation Instructions (VBMI)* – додає векторні інструкції перестановки байтів, яких немає в *AVX-512 BW*;
- *AVX-512 Vector Neural Network Instructions Word variable precision (4VNNIW)* – векторні інструкції для глибокого навчання;
- *AVX-512 Fused Multiply Accumulation Packed Single precision (4FMAPS)* – векторні інструкції для глибокого навчання;
- *VPOPCNTDQ* – підрахунок кількості бітів, встановлених в 1;
- *VPCLMULQDQ* – множення 64-бітних цілих чисел без переносу;
- *AVX-512 Vector Neural Network Instructions (VNNI)*;
- *AVX-512 Galois Field New Instructions (GFNI)* – векторні інструкції для обчислення поля Галуа;
- *AVX-512 векторні інструкції AES (VAES)* – векторні інструкції для кодування AES (*Advanced Encryption Standard*);
- *AVX-512 Vector Byte Manipulation Instructions 2 (VBMI2)* – завантаження байтів/слів, збереження та конкатенація зі зсувом;

- *AVX-512 бітові алгоритми (BITALG)* – розширення інструкцій *VPOPCNTDQ* для підрахунку кількості одиничних бітів в операндах довжиною в байт/слово.

Лише розширення ядра *AVX-512F* вимагається для всіх реалізацій, хоча всі поточні процесори також підтримують *AVX-512CD* (виявлення конфліктів).

Оновлені інструкції *SSE/AVX* у *AVX-512F* використовують ту саму мнемоніку, що й версії *AVX*; вони можуть працювати з 512-бітними регістрами *ZMM*, а також підтримуватимуть 128/256-бітні регістри *XMM/YMM* (з *AVX-512VL*) і цілі операнди довжиною в байт, слово, подвійне слово і зчетверене слово (з *AVX-512BW/DQ* і *VBMI*).

Мікрокроцесори з *AVX-512*: *Xeon Phi x200 (Knights Landing)* (2016), *Knights Mill* (2017), *Skylake-SP*, *Skylake-X* (2017), *Cannon Lake* (2018), *Ice Lake* (2019), *Willow Cove* (2020), *Cypress Cove* (2021), *Golden Cove (2021)*, *Alder Lake (2021*, може бути включеним в BIOS*)*.

В останніх моделях Desktop і Mobile процесорів від Intel *(Alder Lake-2021, Raptor Lake-2022)* впроваджено спрощений варіант *AVX-512,* що має назву *AVX-VNNI* (*Vector Neural Network Instructions* – векторні інструкції для глибокого навчання).

AVX-VNNI забезпечує той самий набір операцій, що і *AVX512-VNNI*, але обмежений 256-бітними векторами та не підтримує жодних додаткових функцій, таких як: трансляція, регістри масок або доступ до більш ніж 16 векторних регістрів. Це розширення дозволяє підтримувати операції *VNNI*, навіть якщо повна підтримка AVX-512 в процесорі не реалізована.

Компілятори, що підтримують *AVX-512*:

- *GCC 4.9* і новіші;
- *Clang 3.9* і новіші;
- *ICC 15.0.1* і новіші;
- *Microsoft Visual Studio 2017 C++* Compiler*;*
- *Java 9.*

На рис. 1.2 показано як *SIMD* команди використовуються для векторізації циклів в програмі. Тут *CPUID* – ідентифікаційкий номер процесора

(шістнадцятирозрядне шістнадцятирічне число), що серед іншого визначає які набори *SIMD* команд підтримує цей процесор.

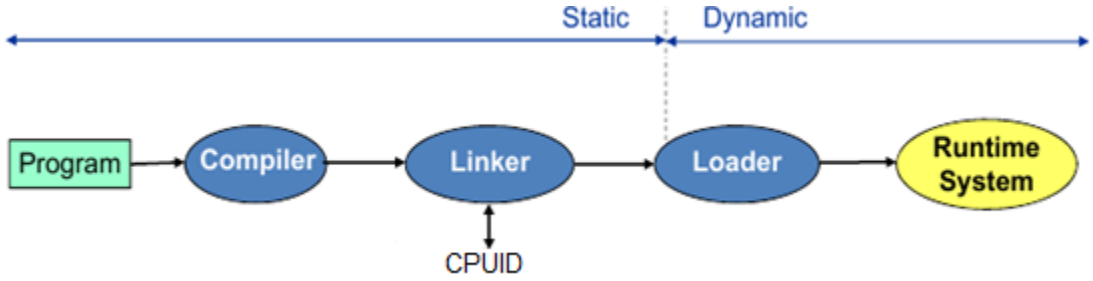

Рисунок 1.2 – Використання *SIMD* для векторизації циклів

Linker (також: компонувальник, редактор зв'язків) – програма, котра виконує компоновку: приймає на вхід один або декілька об'єктних модулів і збирає з них модуль, що буде виконуватися.

Компонувальник може брати об'єкти з колекції, яка називається бібліотекою середовища виконання. Зараз *Intel* і *AMD* надають оптимізовані математичні бібліотеки, які використовують інструкції *SIMD*. Компонувальник також піклується про впорядкування об'єктів в адресному просторі програми. Це може включати переміщення коду, який передбачає певну базову адресу, в іншу базу. Оскільки компілятор рідко знає, де буде знаходитися об'єкт, він часто передбачає фіксоване базове розташування (наприклад, нуль).

Loader (завантажувач) - це частина ОС, яка відповідає за завантаження програм і бібліотек. Це один із важливих етапів у процесі запуску програми, оскільки він розміщує програми в пам'яті та готує їх до виконання. Завантаження програми передбачає читання вмісту виконуваного файлу, що містить інструкції програми, у пам'ять, а потім виконання інших необхідних підготовчих завдань для підготовки виконуваного файлу до запуску. Після завершення завантаження ОС запускає програму, передаючи керування завантаженому програмному коду.

1.3 Паралелізм на рівні команд (*ILP- Instructions-Level Parallelism*)

Комп'ютерна програма – це, по суті, потік інструкцій, що виконується процесором. Без *ILP* ядро процесора може виконувати лише менше однієї

інструкції за такт (*IPC < 1*). Ці процесори відомі як скалярні процесори та субскалярні процесори. Їх також називають процесорами *CISC (complex instruction set computer*- зі складним набором інструкцій*).*

Усі сучасні скалярні та субскалярні процесори мають багатоступеневі конвеєри команд. Кожен етап у конвеєрі відповідає іншій дії, яку процесор виконує над цією інструкцією на цьому етапі; процесор з *N*-ступеневим конвеєром може мати до *N* різних інструкцій на різних стадіях завершення і, таким чином, іноді може видавати майже одну інструкцію за такт [1, 4].

Ядро скалярного процесора може виконувати лише менше однієї інструкції *SISD* (*Single Instructions, Single Data*) за такт (наприклад: *Intel 80486, Intel Pentium*).

Субскалярне ядро процесора *Intel* може виконувати не більше двох інструкцій *SISD* або *SIMD* за такт (наприклад: *Intel Atom, Intel Xeon Phi*, де мікропроцесор *Xeon Phi 7290 («Knights Landing»)* має 72 ядра/288 потоків, але для кожного субскалярного ядра *IPC≤2*. *Bonnell* — це мікроархітектура центрального процесора [5], яка використовується процесорами *Intel Atom* і може виконувати до двох інструкцій за цикл. Як і суперскалярні процесори x86-64, він перетворює інструкції x86-64 (інструкції *CISC*) у простіші внутрішні операції (інструкції *RISC*) перед виконанням. Більшість інструкцій створюють одну *RISC* під час перекладу, приблизно 4% інструкцій, що використовуються в типових програмах, створюють кілька *RISC*. Кількість інструкцій, які створюють більше однієї *RISC*, значно менша, ніж у суперскалярних процесорів. У мікроархітектурі *Bonnell* внутрішні *RISC* можуть містити як завантаження з пам’яті, так і збереження в пам’яті у зв’язку з типом операції *ALU*, таким чином, вони більше схожі на рівень команд *x86*. Це забезпечує відносно хорошу продуктивність лише з двома *ALU* і без будь-якого перевпорядкування інструкцій, спекулятивного виконання чи перейменування регістрів.

Таким чином, мікроархітектура *Bonnell* являє собою часткове відродження принципів, які використовувалися в попередніх розробках *Intel*, таких як *P5*. *Hyper-Threading* (запатентована назва паралелізму на рівні потоків команд від

Intel) реалізовано в *Bonnell* простим (тобто з низьким енергоспоживанням) способом ефективного використання всього конвеєра, з уникненням типових залежностей за даними в одному потоці [5].

Процесор *RISC (reduced instruction set computer* - зі скороченим набором інструкцій) має невеликий набір простих і загальних інструкцій. Головною відмінною рисою *RISC* є те, що набір інструкцій оптимізовано для дуже регулярного конвеєрного потоку інструкцій. Іншою загальною рисою *RISC* є їхня архітектура завантаження/збереження, в якій доступ до пам'яті здійснюється за допомогою спеціальних інструкцій, а не як частини більшості інструкцій. Загальноприйнятним є [1], що продуктивність ядра процесора *RISC* за такт є *IPC=1* (наприклад: *Intel i960, Am29000, ARM, PA-RISC).*

Інструкції можна перевпорядковувати та об'єднувати в групи, які потім виконуються паралельно без зміни результату програми. Це відомо як паралелізм на рівні інструкцій. Алгоритми ***Scoreboarding*** і ***Tomasulo*** (який є подібним до *Scoreboarding*, але використовує перейменування регістрів) є двома найпоширенішими техніками для реалізації позачергового виконання та паралелізму на рівні інструкцій.

1.3.1 Архітектура *VLIW* (*Very Long Instruction Word*)

Архітектура *VLIW* має свій початок від паралельного мікрокоду, що використовувався ще на світанку обчислювальної техніки, і від суперкомп'ютерів *Control Data CDC6600* та *IBM 360/91*.

Першими справжніми *VLIW*-комп'ютерами стали міні-суперкомп'ютери, що були вироблені на початку 1980 року компаніями *MultiFlow, Culler* та *Cydrome*, але вони не мали комерційного успіху. Однак, використовувані в них планувальник обчислень і програмна конвеєризація (розроблені Фішером та Рау (*Cydrom*е)) і сьогодні складають основу технології побудови *VLIW*-компілятора.

Із розробок Радянського Союзу, що внесли вклад в розвиток *VLIW*-архітектури, слід згадати машини Карцева: *М10* та *М13*, а також *«Ельбрус-3».*

Процесор *Intel i860* став першим *VLIW*-процесором на одній мікросхемі. Однак *i860* можливо віднести до *VLIW* достатньо умовно — по суті, у нього є лише програмно-керовані спарені інструкції.

Crusoe від *Transmeta* (Росія, 2000 р.) - всі програми і сама операційна система працюють поверх низькорівневого програмного забезпечення, що називається морфінгом коду (*Code Morphing*) і відповідального за трансляцію *x86*-кодів в зв'язки за розміром *128* біт (молекули), котрі складаються із 4-ьох 32 бітних атомів.

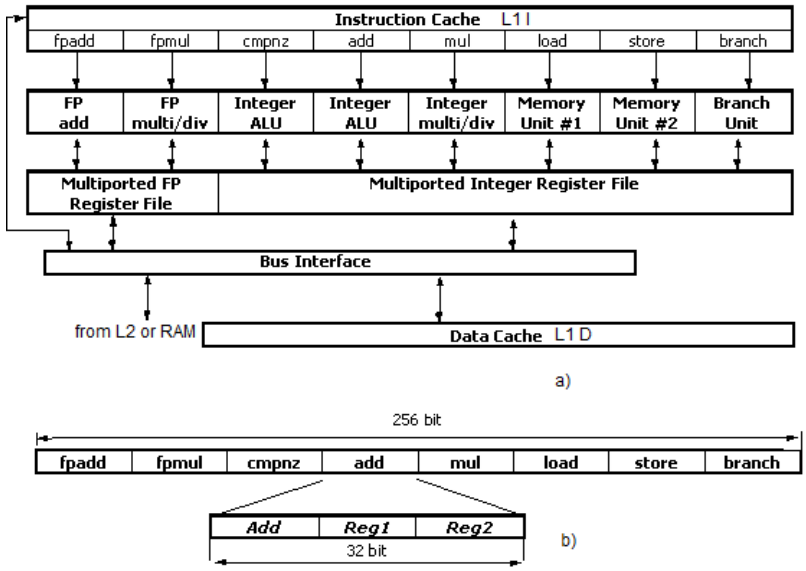

Рисунок 1.3 – Мікропроцесор із *VLIW*-архітектурою

EPIC *(Explicitly Parallel Instruction Computing)* – це еволюція архітектури *VLIW*, котра абсорбувала в себе деякі концепції суперскалярної архітектури, хоча і в формі, адаптованій до *EPIC*. *EPIC* – це «ідеологія», що визначає, як створювати *ILP*-процесори, а також набір характеристик архітектури, котрі підтримують цю ідеологію. Першим прикладом комерційної *EPIC ISA* стала архітектура ***IA-64***. Більш загальну концепцію *EPIC* уявляє собою архітектура ***HPL-PD*** (раніше відома, як архітектура ***Hewlett-Packard Laboratories PlayDoh***).

Дослідження по архітектурі *EPIC* в *HP Labs* проводилися, починаючи з початку *1989* р., хоча сама назва цього типу архітектури з'явилася тільки в *1997*

році, коли був утворений альянс *HP-Intel*. В той час суперскалярні процесори вже були самими популярними засобами досягнення *ILP*.

Розробники цієї архітектури ставили перед собою задачу уникнути позачергового виконання команд – елегантної, але дуже дорогої методики досягнення *ILP* в суперскалярній архітектурі.

Архітектура *VLIW* вирішує задачу досягнення високого рівня *ILP*, знижуючи, при цьому, складність апаратного забезпечення. Однак вона вимагає наявності інтелектуального компілятора, котрому, за звичай, не вдається побудувати ефективний код скалярних додатків з інтенсивним розгалуженням.

На відміну від програм для суперскалярних процесорів, код *VLIW* пропонує точний план того, як процесор буде виконувати програму, план, котрий компілятор створює статично під час компіляції. Код точно вказує, коли буде виконана кожна операція, які функціональні пристрої будуть працювати і які регістри будуть містити операнди. Компілятор *VLIW* створює такий план виконання (***plan of execution – POE***), маючи повне уявлення про процесор *VLIW*, причому створює цей план так, щоб досягти потрібного запису виконання (***record of execution, ROE***) – послідовності подій, котрі дійсно відбуваються під час роботи програми.

Одна із цілей, котрі розробники ставили перед собою при створенні EPIC, полягала в тому, щоб зберегти реалізований у VLIW принцип статичного створення POE, але в той же час збагатити його можливостями, подібними до можливостей суперскалярного процесора, дозволяючи новій архітектурі краще враховувати динамічні фактори, що традиційно обмежують паралелізм, властивий VLIW. Щоб добитися цих цілей, «ідеологія» *EPIC* була побудована на деяких головних принципах.

Перший — це створення плану виконання під час компіляції. *EPIC* кладе тягар по створенню *POE* на компілятор. Хоча, взагалі, архітектура і фізична реалізація можуть перешкоджати компілятору в виконанні цієї задачі, процесори *EPIC* надають функцій, котрі допомагають компілятору створювати план виконання. Під час виконання поведінка процесора *EPIC,* з точки зору

компілятора, повинна бути передбачуваною і керованою. Динамічне позачергове виконання команд може «заплутати» компілятор так, що він не буде «розуміти», як його рішення вплинуть на реальний запис виконання, створений процесором, тому йому необхідно вміти передбачати дії процесора, що ще більше ускладнює задачу.

Суть створення плану під час компіляції *полягає в переупорядкуванні початкового послідовного коду таким чином, щоб використати всі переваги паралелізму додатка і максимально ефективно тратити апаратні ресурси, мінімізуючи час виконання.* Без відповідної підтримки архітектури таке переупорядкування може порушити коректність програми. Таким чином, оскільки *EPIC* покладає створення *POE* на компілятор, вона повинна забезпечити ще і архітектурні можливості, підтримуючі інтенсивне переупорядкування коду під час компіляції.

Наступним принципом є використання компілятором імовірнісних оцінок. Компілятор *EPIC* зштовхується з значною проблемою при створенні плану виконання, т. щ. інформація, котра дуже впливає на запис виконання, становиться відомою тільки лише в момент виконання програми. Наприклад, компілятор не може точно знати, котра з гілок програми буде виконуватися, після виконання умовного оператора. Це стане відомим, коли умова буде обрахована і проаналізована. Крім того, зазвичай неможливо створити статичний план, котрий одночасно оптимізує всі шляхи в програмі після розгалужень.

Одним з найважливіших принципів *EPIC* є дозвіл компілятору оперувати імовірними оцінками. Компілятор створює і оптимізує *POE* для найбільш імовірних подій.

EPIC забезпечує також і апаратну підтримку виконання *POE*, таку як спекулятивне виконання по керуванню і по даним (***Control and Data Speculation***). Ця апаратна підтримка забезпечує коректність програми, навіть якщо початкові припущення при складанні *POE* були не вірними. Однак, коли початкове припущення виявляється невірним, відбувається падіння продуктивності при виконанні програми.

В прикладі фрагмента програми: *A=B+C*; *B=D+E*. Значення змінної *A* буде різним в залежності від порядку, в котрому виконуються ці оператори (спочатку обчислюється *A*, а потом *B*, чи навпаки), але ж програміст при написанні програми мав чітко виражений порядок їх обчислення. При паралельному обчисленні цих операторів, на вірний результат можливо розраховувати лише з деякою вірогідністю, а не гарантовано.

В архітектурі *EPIC,* також як і в архітектурі *VLIW,* явний паралелізм представлений на рівні команд, котрі керують одночасною роботою функціональних виконуючих пристроїв (ФП).

В *IA-64* зв'язка має довжину *128* біт. Вона включає в себе *3* поля для команд довжиною *41* біт кожне (слоти) і *5*-розрядний шаблон (Mask). Команди зв'язки можуть виконуватися паралельно різними ФП. Можливі залежності, перешкоджаючі паралельному виконанню команд однієї зв'язки, відображаються в полі шаблону. *Паралельно можуть виконуватися і команди 2-3 сусідніх зв'язок.*

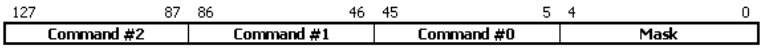

Рисунок 1.4 – Наддовге командне слово (зв'язка) в архітектурі *IA-64*

Шаблон показує, якого типу є команди, що знаходяться в слотах зв'язки. В загальному випадку однотипні команди можуть виконуватися в більш ніж одному типі ФП. Шаблоном задаються також зупинки, що визначають лот, після початку виконання команди котрого, інструкції наступних спотів повинні ждати завершення виконання команди попереднього слову.

Intel, починаючи з процесорів *Itanium 2,* зберегла можливість виконання в архітектурі *IA-64* класичного *x86*-коду, правда, вдалось їй це зробити ціною колосальних втрат продуктивності при роботі процесора в такому режимі.

Історія розвитку архітектури I*A-64*: *Itanium (Merced),* 2001, 1 ядро, 180 нм => *Itanium2 (McKinley* and *Madison),* 2002–2005, 1 ядро, 130 нм =>*Itanium 2 9000*

і *Itanium 9100 (McKinley* і *Madison)*, 2006–2007, 1 або 2 ядра, 90 нм =>*Itanium 9300 (Tukwila)*, 2010, 2 або 4 ядра, 65 нм => *Itanium 9500 (Poulson)* обробка до 12 команд за один такт, удосконалена багатопоточність та нові інструкції для використання переваг паралелізму, особливо у віртуалізації, 2012, 4 або 8 ядер, 32 нм =>*Itanium 9700 (Kittson)* не мав покращень мікроархітектури порівняно з *Poulson* (збільшено лише тактову частоту та розмір кешу *L3*), 2017, 4 або 8 ядер, 32 нм.

1.3.2 Знайомство із суперскалярною архітектурою

Винахідник *RISC – David Patterson* (Berkeley, California в середині 70-х). Винахідник *CISC-RISC «CISC-outside RISC-inside» – Yale Patt* (Berkeley, California, 1985) [6], де вона описана як *High Performance Substrate (HPS). HPS* – це нова мікроархітектура, що призначена для впровадження в дуже високопродуктивних обчислювальних механізмах. ЇЇ модель виконання – це обмеження на потік даних із дрібною деталізацією *(RDF - restriction on fine granularity data flow).* У статті [6] представлено модель, надано обґрунтування її вибору та описано шлях даних і потік інструкцій через мікроінструмент. По суті в [6] здійснено розвиток архітектури *Dataflow* на рівні *CISC* до *Restricted Dataflow* на рівні *RISC.*

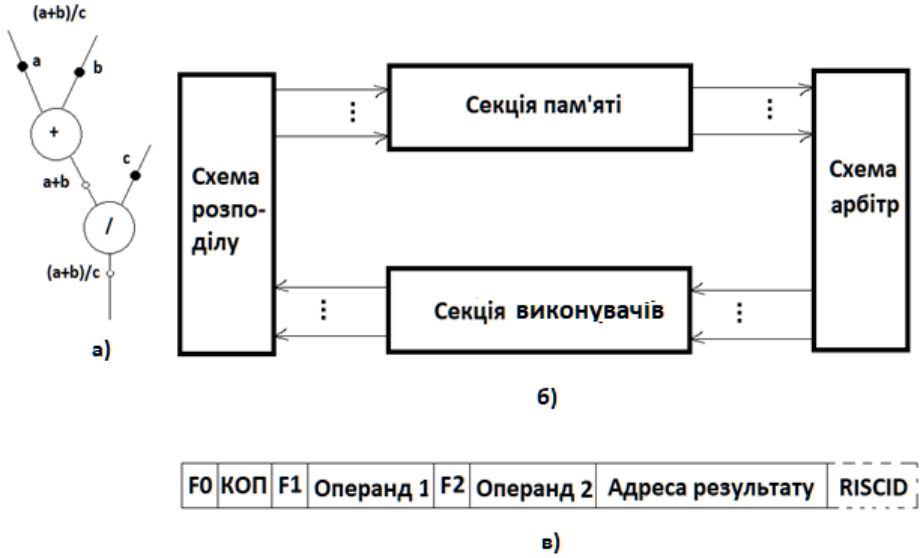

Рисунок 1.5 – Архітектура *Dataflow*

Винахідник архітектури *Dataflow* (кінець 70-х, початок 80-х) – *Jack Dennis* із MIT (див. рис. 1.5). Відповідно до цієї архітектури вся обчислювальна програма представляється схемою потоку даних і складається з арифметичних та логічних акторів. На рис. 1.5. а) показаний фрагмент програми для обчислення арифметичного виразу *(a+b)/c*, де арифметичні актори представлені кругами, що мають по одній вхідній дугі для кожного операнду та вихідну дугу для результату виконання актора. Секція пам'яті (див. рис. 1.5. б) складається з активних комірок пам'яті, що мають структуру з рис. 1.5. в) (фрагмент виділений пунктиром – це доповнення *Yale Patt*).

Кожному акторові схеми потоку даних відповідає активна комірка пам'яті в секції пам'яті. При надходженні операнду на вхідну дугу актора (чорна точка на рис. 1.5. а)) схема розподілу з рис. 1.5. б) записує його в активну комірку пам'яті, що відповідає цьому акторові. Паралельно з цим в комірці значення одного з прапорців Fi змінюється на 1. Коли актором отримані всі операнди, з якими він працює, в відповідній активній комірці пам'яті всі очікувані прапорці Fi будуть встановлені в 1. При цьому, у зв'язку з тим, що кожна комірка наділена не складною логічною схемою в ній автоматично змінюється на 1 значення прапорця F0. Ця подія інформує схему арбітр з рис. 1.5. б), що відповідний актор схеми потоку даних готовий до виконання і якщо на цей момент часу в секції виконувачів є вільним виконуючий пристрій, що може виконати потрібну операціє, то її код, разом з операндами і адресом комірки пам'яті (куди потрібно записати результат) можуть бути переданими на виконання в цей виконуючий пристрій.

Таким чином, в архітектурі *Dataflow* здійснюється автоматичне розгалуження готових до виконання акторів схеми потоку даних на множину виконуючих пристроїв і при цьому відпадає необхідність наявності лічильника команд.

Було виконано багато проектів, що намагалися реалізувати наближення до архітектури *Dataflow*, розробленої *Jack Dennis,* але всі вони мали проблеми із складністю її реалізації у зв'язку з: великою кількістю можливих типів акторів в програмі; складністю реалізації секції пам'яті і усуненню залежностей за даними

між готовими до виконання комірками пам'яті. Все це мало місце поки *Yale Patt* не прийшла ідея обмежити *Dataflow* до *RDF*, перейшовши від можливих типів акторів в схемі потоку даних від *CISC* до *RISC*. При цьому відпала необхідність в наявності великої кількості активних комірок пам'яті і спростилась проблема усунення проблем залежностей за даними (проблема отримання результату паралельного виконання фрагменту програми, котрий би відповідав її послідовному виконанню – так як це бачив програміст).

Intel P6 (1995) - перша суперскалярна мікроархітектура. В ній реалізована технологія *OoOE* (*Out-of-Order Execution*), що заснована на використанні *RDF*.

Інші суперскалярні мікроархітектури, що також засновані на використанні *RDF*:

- мікроархітектура невеликої приватної компанії *NexGen* (в 1996 р. придбана компанією *AmD*, з 1997 р. за цією мікроархітектурою випускалися мікропроцесори *AmD K6*);
- мікроархітектура невеликої приватної компанії *Cyrix* (в 1997 р. придбана компанією *National Semiconductor*, котра в 1999 р. була поглинута Тайванським підприємством *VIA Technologies*, із 1999 р. за цією мікроархітектурою випускалися мікропроцесори, спочатку під назвою *Cyrix*, а пізніше під назвою *VIA*).

1.3.3 Моделі розвитку мікропроцесорів *Intel* з архітектурою *CISC-RISC*

Модель ***tick-tock*** – екстенсивна стратегія розробки мікропроцесорів, що анонсована *Intel* на конференції *Intel Developer Forum* у вересні 2006. Цикл розробки ділиться на дві стадії – *tick* і *tock*: *tick* значить мініатюризацію технологічного процесу та відносно невеликі покращення мікроархітектури; *tock* значить випуск процесорів з новою мікроархітектурою, але на основі вже відпрацьованого технологічного процесу. По планам *Intel*, кожна частина циклу повинна була займати приблизно рік (див. табл.1.1). Починаючи з 2016 р. *Intel* перейшла на нову модель ***process–architecture–optimization*** (див.: табл.1.1 і 1.2)**.**

Таблиця 1.1 – Основні етапи розвитку *CISC-RISC* архітектури мікропроцесорів фірми *Intel (x86-64).*

Стадія	Мікроархітектура	Технологічний процес	Початок випуску
1	2	3	4
tock	*Core*	65 нм	2006
tick	*Penryn* (покращена *Core*)	45 нм	2008
tock	*Nehalem*		2009
tick	*Westmere* (покращена *Nehalem*)	32 нм	2010
tock	*Sandy Bridge*		2011
tick	*Ivy Bridge* (покращена *Sandy Bridge*)	22 нм	2012
tock	*Haswell*		2013
tick	*Broadwell* (покращена *Haswell*)	14 нм	2014
tock (7th-generation)	*Skylake*		2015
Optimization	*Kaby Lake*		2016
optimization	*Coffee Lake*		2017
optimization	*Whiskey Lake*		2018
optimization	*Comet Lake*		2019
process	*Cannon Lake*	10 нм	2018 - годні тільки *i3* в невеликій кількості
architecture (10th-generation)	*Sunny Cove* (мобільна версія *Ice Lake)*		2019

optimization	*Tiger Lake*		2020
optimization (11th-generation)	*Willow Cove (Willow Cove is almost identical to the previous microarchitecture but introduces new security features, a redesigned cache subsystem, and higher clock speeds. Intel claims that these changes, in addition to the new 10SF process node, give an additional 10–20% performance increase from Sunny Cove)*		09.2020
optimization	*Cypress Cove*	14 нм	11.2021
optimization	*Golden Cove*	*10 нм*	11.2021
optimization (12th-generation)	*Alder Lake (Hybrid architecture utilizing Golden Cove high-performance cores and Gracemont (SoC, 4th-generation out-of-order low-power Atom microarchitecture) power-efficient cores, up to 16 (8 P-cores+8 E-cores))*		11.2021
optimization (13th-generation)	*Raptor Lake (Up to 8 Raptor Cove performance cores (P-core); up to 16 Gracemont efficient cores (E-core))*		10.2022

Таблиця 1.2 – Покращення ключових параметрів *CISC-RISC* архітектури мікропроцесорів фірми *Intel*.

	Sandy Bridge	Haswell	SkyLake	Sunny Cove
Out-of-order Window	168	192 ↑	224 ↑	352 ↑
In-flight Loads	64	72 ↑	72	128 ↑
In-flight Stores	36	42 ↑	56 ↑	72 ↑
Scheduler Entries	54	60 ↑	97 ↑	160 ↑
Integer Register File	160	168 ↑	180 ↑	
FP Register File	144	168 ↑	168	
Allocation Queue	28/thread	56 or 28/thread	64/thread ↑	64/thread

1.4 Паралелізм на рівні потоків команд

Intel використовує термін «потік» як загальний термін для термінів «процес» і «потік» [2].

Паралелізм на рівні потоків команд в процесорах *Intel* був впроваджений ще в процесорах з архітектурою *IA-32 (x86)*, але вже після того як ця архітектура набула риси *CISC-RISC*. Реалізація цього паралелізму отримала запатентовану назву технології *Intel Hyper-Threading (Intel HT Technology)*. Вона була розроблена для підвищення продуктивності процесорів під час виконання багатопотокової операційної системи та коду програми або однопотокових програм у багатозадачному середовищі. Технологія дозволяє одному фізичному процесору виконувати два або більше окремих потоків коду одночасно, використовуючи спільні ресурси виконання.

Технологія *Intel HT* забезпечує апаратну багатопотоковість на одному фізичному пакеті за допомогою наявності спільних ресурсів виконання в ядрі процесора для різних потоків команд. Архітектурно процесор, який підтримує технологію *Intel HT*, складається з двох або більше логічних процесорів, кожен з яких має власний архітектурний стан (*AS*). Кожен логічний процесор складається з повного набору архітектурних регістрів. Кожен також має свій розширений програмований контролер переривань (*APIC*).

На рис. 1.6 показано порівняння процесора, який підтримує технологію *Intel HT* (реалізований з двома логічними процесорами) і традиційної двопроцесорної

системи.

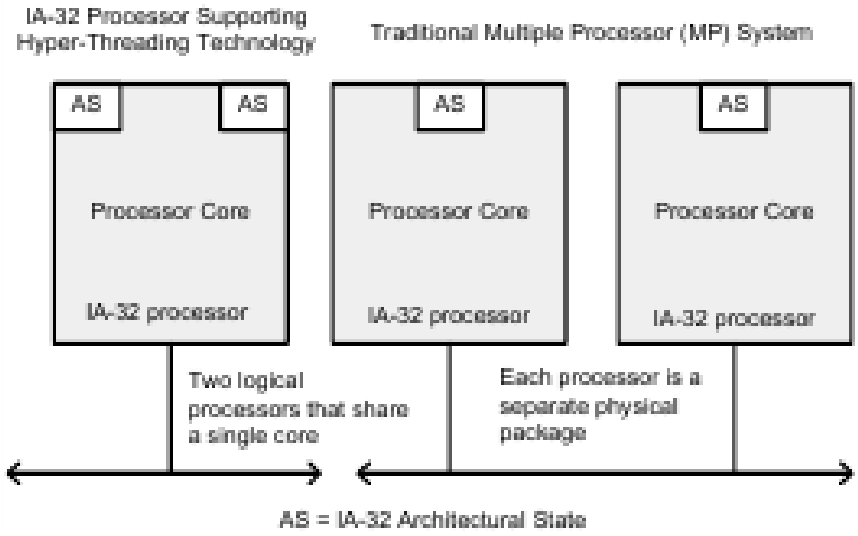

Рисунок 1.6 − Порівняння процесора *IA-32* із підтримкою технології *Hyper-Threading* і традиційної двопроцесорної системи.

На відміну від традиційної системної конфігурації *MP*, яка використовує два або більше окремих фізичних процесорів *IA-32*, логічні процесори в процесорі *IA-32*, що підтримує технологію *Intel HT*, спільно використовують основні ресурси фізичного процесора. Це включає механізм виконання та інтерфейс системної шини. **Після ввімкнення живлення та ініціалізації кожен логічний процесор може бути незалежно спрямований на виконання вказаного потоку команд, переривання чи зупинку**.

Технологія *Intel HT* використовує з вигодою паралелізм процесів і потоків, який можна знайти в сучасних операційних системах і високопродуктивних програмах, надаючи два або більше логічних процесорів на одному чіпі. Ця конфігурація дозволяє виконувати два або більше потоків одночасно на кожному фізичному процесорі. Кожен логічний процесор виконує інструкції з потоку програми, використовуючи ресурси в ядрі процесора. Ядро виконує ці потоки одночасно, використовуючи планування інструкцій поза порядком (*out-of-order instruction scheduling*), щоб максимізувати використання блоків виконання протягом кожного такту.

Усі конфігурації технології *Intel HT* вимагають [7]:

• Процесор, що підтримує технологію *Intel HT*;

• Чипсет і BIOS, які використовують цю технологію;

• Оптимізацію операційної системи.

На рівні вбудованого програмного забезпечення (*BIOS*) основні процедури ініціалізації логічних процесорів у процесорі, що підтримує технологію *Intel HT*, такі ж, як і для традиційної платформи *DP* (двопроцесорна) або *MP* (багатопроцесорна). Механізми, описані в специфікації багатопроцесорного процесора версії 1.4 для ввімкнення та ініціалізації фізичних процесорів у системі *MP*, також застосовуються до логічних процесорів у процесорі, який підтримує технологію *Intel HT*. Операційна система, що призначена для роботи на традиційній платформі *DP* або *MP*, може використовувати *CPUID* для визначення наявності апаратної функції багатопоточної підтримки та кількості логічних процесорів, які вони забезпечують. В *Atom* і *Xeon Phi (Knights Landing-2016, Knights Mill -2017) Hyper-Threading* реалізовано простим (тобто з низьким енергоспоживанням) способом, щоб ефективно використовувати весь конвеєр, уникаючи типових залежностей одного потоку.

1.5 Паралелізм на рівні файберів

Крім терміну «потік» *Intel* також використовує і більш загальний термін «*task».* При цьому *task* (завдання) розглядається як одиниця роботи, котру процесор може відправляти на виконання, виконувати та призупиняти [*2*]. Управління завданнями та перемикання має здійснюватися програмним забезпеченням.

Всі виконання в процесорі відбуваються в рамках завдання. Навіть прості системи повинні визначати хоча б одне завдання. Більш складні системи можуть використовувати засоби керування завданнями процесора для підтримки програм з багатьма завданнями.

Tasks-Level Parallelism - це характеристика паралельної програми, яка полягає в тому, що "можуть виконуватися абсолютно різні обчислення на однакових або різних наборах даних". Це контрастує з паралелізмом даних, коли ті самі

обчислення виконуються на однакових або різних наборах даних. Паралелізм завдань передбачає декомпозицію завдання на підзавдання, а потім виділення кожному такому завданню логічного процесору для виконання. При цьому, логічні процесори виконують ці завдання паралельно (***concurrently***) на різних логічних процесорах і часто змінюючи одна одну на одному і тому ж логічному процесорі (***cooperatively***). Паралелізм завдань зазвичай не залежить від розміру проблеми.

Fiber (волокно) є особливо легким потоком виконання. Як і потоки, волокна спільно використовують адресний простір. Однак волокна використовують кооперативну багатозадачність (***cooperative multitasking***), тоді як потоки використовують багатозадачність з пріоритетним перериванням ***(preemptive multitasking)***.

До .NET Framework була ***множина різних (і часто несумісних між собою) середовищ і технологій програмування***, у зв'язку з тим що розробка інструментів для програмістів була мовно орієнтованою: для *Visual Basic* існував свій набір додатків, для *C++* – свій, для *Java* – свій.

В 2002 р. *Microsoft* випустила об'єднання всіх найбільш вдалих уніфікованих напрацювань в рамках єдиної платформи *.NET Framework 1.0.* В рамках *.NET Framework,* ставиться завдання відповідності всім актуальним тенденціям в галузі програмування на теперішній момент часу. Так, наприклад, *.NET Framework* напряму підтримує об'єктно-орієнтованість, безпеку типів, «прибирання сміття», структурну обробку виключень і т. п.

Однією з наробок, що входять в *NET Framework* і використовують паралелізм на рівні *fiber* в рамках *threads,* є ***Thread pool***. *Thread pool* (див рис. 1.7) зменшує витрати на створення нового потоку шляхом повторного використання обмеженої кількості потоків.

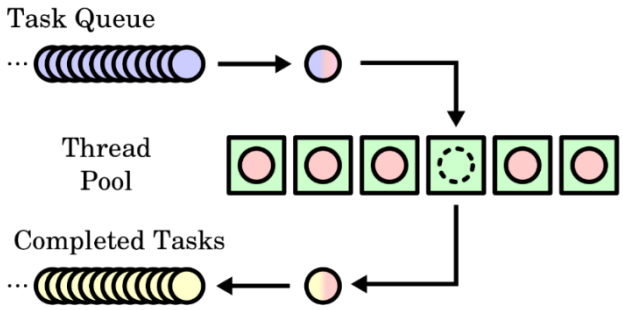

Рисунок 1.7 – *Thread pool*

В серпні 2022 *Microsoft* випустила версію *.NET Framework 4.8.1*. *.NET Framework* – це запатентована програмна основа (*software framework*), яка в основному працює в *Microsoft Windows*. У 2014 році *Microsoft* анонсувала *.NET Core*, намагаючись включити крос-платформну підтримку для *.NET*, включаючи *Linux* і *macOS*. *.NET Core* – це крос-платформний наступник *.NET Framework*. Проект в основному розроблено співробітниками *Microsoft* за допомогою *.NET Foundation* і випущено за ліцензією MIT. В 2020 р. нова версія *.NET Core* вийшла під назвою *.NET5*. В 2022 р. вийшла версія *.NET7* (підтримується *Visual Studio 2022 Version 17.4*).

1.6 Архітектура *x86-64*

Архітектура ***x86-64*** (також *AMD64/x64/Intel64/IA-32e/EM64T*, але не *IA-64*) – **64-бітна апаратна платформа (система команд, архітектура мікропроцесора та чипсет)**, розроблена компанією *AMD* для виконання 64-разрядних додатків. Це 64-бітове розширення архітектури *x86* з майже повною зворотною сумісністю.

Архітектура *x86-64* підтримує значно більшу кількість віртуальної пам’яті додатків і фізичної пам’яті комп’ютера, ніж архітектура *x86*, дозволяючи додаткам користувачів зберігати значно більшу кількість даних в пам’яті. Архітектура *x86-64* також забезпечує 64-бітні універсальні регістрі та численні інші розширення. Оригінальну специфікацію цієї архітектури було створено *AMD* та реалізовано *AMD*, *Intel*, *VIA* і іншими. Вона є повністю сумісною з 16-

розрядним і 32-розрядним кодом *x86*. Оскільки повні *x86* 16-розрядні та 32-разрядні набори команд повністю реалізуються апаратними засобами цієї архітектури і не потребують ніякої додаткової емуляції, існуючі *x86* виконувані файли виконуються на ній без яких-небудь втрат продуктивності. Якщо існуючі додатки закодувати повторно, щоб використати в своїх інтересах нові особливості архітектури *x86-64*, то можливо досягнути значного підвищення продуктивності.

Мікроархітектура процесорів *AMD K8* (була анонсована в 1999 р., перші процесор з'явилися в 2003 р.) стала першою реалізацією архітектури *x86-64*. При цьому в *AMD K8* поняття *x86-64* мало назву ***AMD64*** та **використовувалось для визначення розширення *x86*: набора команд, набора регістрів, обсягу віртуальної пам'яті додатків, обсягу фізичної пам'яті комп'ютера.**

В теперішньому часі найчастіше розширення *x86* називається, як *x86-64*. *Intel* спочатку використовувала для них назву: *IA-32e* та *EM64T*. *Sun Microsystems* (тепер *Oracle Corporation*) та *Microsoft*, використовують *x64*, в той час як *BSD OSs* та *Linux* використовують *AMD64*.

Найбільш значні зміни *x86-64*, по відношенню до *x86*, включають:

- **можливість обробки 64-бітових цілочислових обчислень:** Всі регістри загального призначення (*GPRs*) розширені від 32 бітів до 64 бітів. При цьому всі арифметичні та логічні операції між: регістрами; пам'яттю і регістром; регістром і пам'яттю тепер можуть безпосередньо обробляти 64-бітові цілі числа. В стек (індексні регістри – див. рис. 1.8) можуть розміщуватися і виштовхуватися з нього дані шириною 8 байт. Покажчик команди може мати ширину 8 байт.

AMD64 регістри

$^{6}_{3}$ ... $^{4}_{7}$ ... $^{3}_{1}$ ... $^{1}_{5}$ ... $^{0}_{7}$ ... $^{0}_{0}$ *(bit position)*

Основні регістри *(8/16/32/64 bits)*

RAX	EAX	AX	AL	**A** register
RBX	EBX	BX	BL	**B** register

RCX	ECX	CX	CL	**C** register
RDX	EDX	DX	DL	**D** register

Індексні регістри *(16/32/64 bits)*

RSI	ESI	SI	**S**ource **I**ndex
RDI	EDI	DI	**D**estination **I**ndex
RBP	EBP	BP	**B**ase **P**ointer
RSP	ESP	SP	**S**tack **P**ointer

Додаткові регістри *(64 bits)*

R8	Register 8
R9	Register 9
R10	Register 10
R11	Register 11
R12	Register 12
R13	Register 13
R14	Register 14
R15	Register 15

Покажчик команд *16/32/64 bits)*

RIP	EIP	IP	**I**nstruction **P**ointer

Сегментні регістри *(16 bits)*

CS	**C**ode **S**egment
DS	**D**ata **S**egment
ES	**E**xtra **S**egment
FS	**F S**egment
GS	**G S**egment
SS	**S**tack **S**egment

ZMM (AVX 512/AVX2/AVX/SSE/MMX) векторні регістри *(512 bits)*

ZMM0 *(512 bits)*	Register 0
ZMM1 *(512 bits)*	Register 1

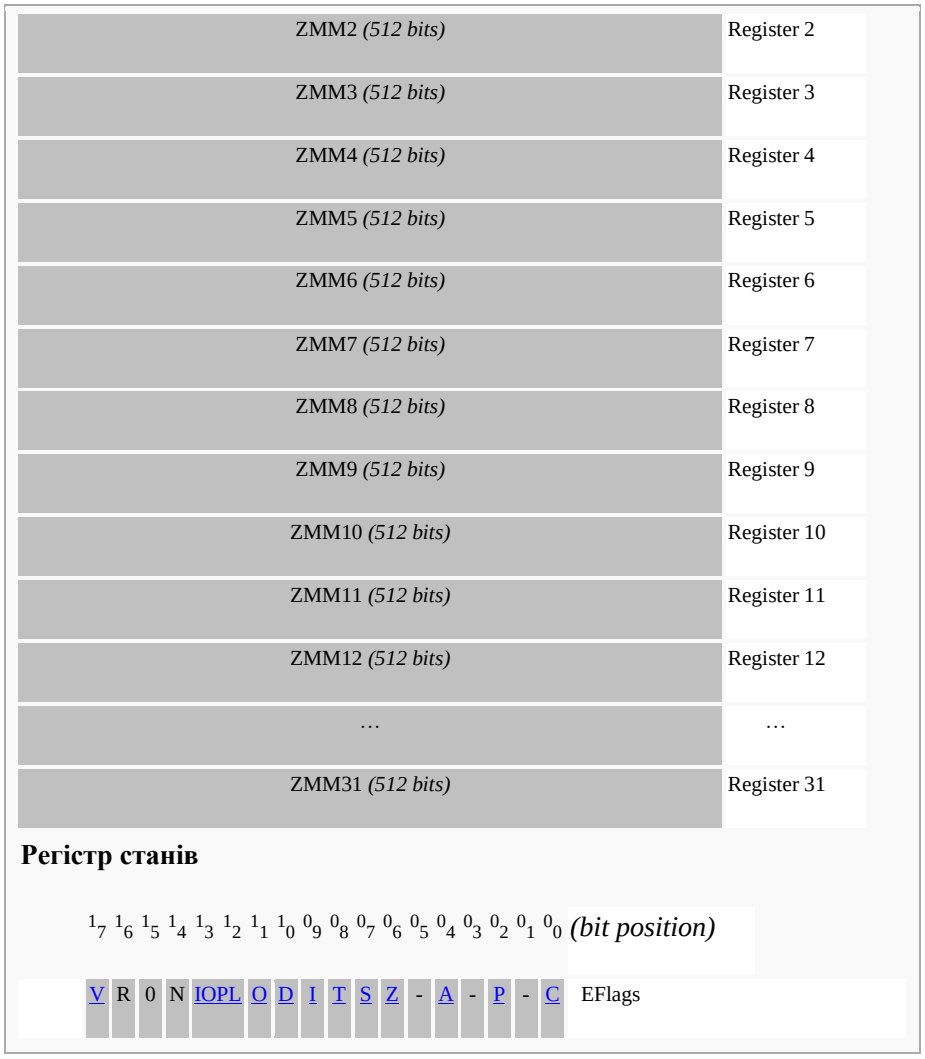

Рисунок 1.8 – Регістри в архітектурі *x86-64*

- **Додаткові скалярні регістри:** На додаток до збільшення розміру регістрів загального призначення (скалярних регістрів), їх кількість збільшилась від восьми (*EAX*, *EBX*, *ECX*, *EDX*, *EBP*, *ESP*, *ESI*, *EDI*) в *x86* до 16 (*RAX*, *RBX*, *RCX*, *RDX*, *RBP*, *RSP*, *RSI*, *RDI*, *R8*, *R9*, *R10*, *R11*, *R12*, *R13*, *R14*, *R15*). Це дозволяє зберігати більшу кількість локальних змінних в регістрах, а не в стеку. Константи, що часто використовуються, тепер також можуть зберігатися в регістрах. В більшості випадків через регістри тепер стало можливим виконання швидкої передачі параметрів.

Однак, у *x86-64* все ще остається значно меншою архітектурна кількість регістрів загального призначення, чим у багатьох *RISC*-архітектур (у котрих зазвичай є 32–64 таких регістрів), або у *VLIW*-подібних архітектур, таких як *IA-64* (архітектура *EPIC*, у котрої 128 регістрів). В архітектурі *CISC-RISC*, котра

реалізує розширення *x86-64*, використовується механізм перейменування регістрів с числом фізичних регістрів, набагато перевищуючим число архітектурних регістрів.

Приклад використання основних, індексних, додаткових регістрів і стеку для передачі аргументів функції та розміщення результатів обчислень, у відповідності до вимог *System V AMD64 ABI,* приведений в [8].

- **Додаткові векторні регістри:** число 128-бітовых регістрів *XMM* (в наступному 256-бітових регістрів *YMM,* а ще пізніше 512-бітових регістрів *ZMM*), що використовуються при виконанні векторних команд *SIMD* також збільшилася від 8 до 16. В нових мікропроцесорах, що підтримують набір команд *AVX-512,* вони стали 512-бітними, а їх кількість, в деяких моделях, збільшилась до 32.
- **Більший віртуальний адресний простір**: архітектура *x86-64* визначає 64-бітовий формат для адресації віртуальної пам'яті додатків. У поточних реалізаціях *x86-64* для цієї мети використовуються тільки 57 молодших бітів. Це дозволяє адресувати віртуальну пам'ять до 128 *PB*. Обсяг віртуальної пам'яті додатків в архітектурі *x86* не міг перевищувати 4 *GB*.
- **Більший фізичний адресний простір:** оригінальна реалізація архітектури *AMD64* здійснила 40-бітові фізичні адреси та дозволила звертатися додаткам до *RAM* з ємністю в 1 *TB*. Більш пізніші реалізації архітектури *x86-64*, починаючи з мікроархітектури *AMD K10*, розширили фізичний адресний простір до 48-біт. Довгий час ядра процесорів з архітектурою *x86-64* мали теоретичний поріг *RAM* 256 *TB*. В 2019 р., в архітектурі *Intel Sunny Cove*, цей поріг був розширеним до 52 біт, що обмежується форматом входу таблиці сторінок. Теоретичний простір оперативної пам'яті в комп'ютері, починаючи з 2019 р., може складати 4 *PB*. Для порівняння 32-разрядні *x86* процесори були обмежені 64 *GB RAM* при реалізації режиму фізичного розширення адреса (*PAE*), або 4 *GB* - без режиму *PAE*. Це означає, що додатки користувачів тепер можуть працювати з дуже великими файлами, відображуючи їх цілком в *RAM*, і не витрачати час на підкачку потрібних сторінок віртуальної пам'яті комп'ютера з *HDD* чи *SDD* в *RAM*.

- **Більший фізичний адресний простір в успадкованому режимі:** працюючи в успадкованому режимі (***legacy***) архітектура *AMD64* підтримує режим (*PAE*) розширення адреса, також, як і найбільш продвинуті *x86* процесори, але *AMD64* розширює *PAE* від 36 бітів (у *x86*) до архітектурного порога 52 бітів фізичної адреси.
- **Доступ до даних відносно покажчика команди:** Команди можуть тепер посилатися на дані відносно покажчика команди (регістр *RIP*). Такі посилання часто використовуються при звертанні до загальнодоступних бібліотек і коду, що загружається під час виконання.
- ***No-Execute* біт:** біт "*NX*" *(No eXecute)* – в термінології *AMD, Intel* називає цей атрибут *XD*-біт (*eXecution Disable*). Це – 63-ій біт із входу таблиці сторінок. Використовуючи його, операційна система (ОС) визначає, які сторінки віртуального адресного простору додатка можуть містити виконуваний код, а які не можуть. Це дозволяє забезпечити більш надійний захист системи від програмних помилок, а також від вірусів, троянських коней та інших шкідливих програм. Сторінка, що помічена атрибутом *NX,* може бути використана для зберігання даних, але не програмного коду. При спробі передати керування на таку сторінку процесор сформує особливий випадок помилки сторінки і програма (частіше за все) буде завершена аварійно. Атрибут захисту від виконання давно був присутнім в інших мікропроцесорних архітектурах, однак в мікропроцесорах із родини *x86* такий захист реалізовувався тільки на рівні програмних сегментів, механізм котрий давно не використовується сучасними ОС. Тепер він добавлений ще й на рівні окремих сторінок.

Сучасні ОС поділяють програми користувачів на сторінки коду *("text"),* даних *("data")*, неініційованих даних *("bss")*, а також **динамічно поділювану область пам'яті**, котра складається з **кучі** ("*heap*", див. далі) та **програмного стеку** *("stack")*. Якщо програма написана без помилок, покажчик команд ніколи не вийде за границі сторінки коду, однак, в результаті програмних помилок, керування може бути передане в інші області пам'яті. При цьому процесор перестане виконувати які-небудь запрограмовані дії, а буде виконувати

випадкову послідовність команд, за котрі він буде приймати дані, що зберігаються на цій сторінці, до тих пір, поки не зустріне недопустиму послідовність, або спробує виконати операцію, що порушує цілісність системи, котра викличе спрацювання системи захисту. В обох випадках програма завершиться аварійно. Також процесор може зустріти бітову послідовність, що інтерпретується як команда переходу до вже пройденої адреси. В такому випадку процесор увійде в безкінечний цикл і програма "зависне".

Основним мотивом введення такого атрибуту було те, що дуже часто такі помилки використовувались зловмисниками для отримання несанкціонованого доступу до комп'ютера, а також написання вірусів. З'явилась велика кількість таких вірусів та черв'яків, що використовували вразливості в поширених програмах.

Один із сценаріїв таких атак полягає в тому, щоб скористувавшись переповненням буфера в програмі (часто це демон, що надає деякий мережевий сервіс), записати спеціальний код в область даних вразливої програми. В результаті переповнення буферу цей код отримає керування і виконає дії, запрограмовані зловмисником (часто це запит виконати програму-оболонку ОС, за допомогою якої зловмисник отримає контроль над системою з правами власника вразливої програми, часто це *root).*

Переповнення буферу часто виникає, коли розробник програми виділяє деяку область даних (буфер) фіксованої довжини, рахуючи, що цього буде досить, а потім не провіряє можливість виходу за границі цієї області. В результаті дані, що надходять, займуть області пам'яті їм не призначені, знищивши інформацію, що там знаходилася раніше. Дуже часто тимчасові буфери виділяються всередині методів, пам'ять для котрих виділяється в програмному стеку, в котрому також зберігаються адреси повернення в викликаючі методи. Ретельно вивчивши код такої програми, зловмисник може виявити таку помилку, і тепер йому досить передати в програму таку послідовність даних, обробивши котру програма помилково замінить адресу повертання в стеку на адресу, потрібну зловмиснику, котрий також передав під виглядом даних деякий програмний код.

Після завершення методу інструкція повертання (*RET*) виштовхне із стека в покажчик команд адрес входу в код зловмисника. Контроль над комп'ютером отриманий.

Завдяки атрибуту *NX (XD)* такий несанкціонований доступ до комп'ютера становиться неможливим. Область стека помічається *NX*-бітом і любе виконання коду в ньому є забороненим. Тепер, якщо передати керування стеку, то спрацює захист. Хоча програму й можливо заставити аварійно завершитися, але багато шкоди це не принесе.

Біт *NX* є самим старшим розрядом елемента 64-бітних таблиць сторінок. Ці таблиці сторінок використовуються операційними системами, що працюють в 64-бітному режимі, або з включеним розширенням фізичних адрес (*PAE*). ***Якщо ОС використовує 32-розрядні таблиці сторінок, то можливості використовувати захисту сторінок від виконання немає***.

Сегментовану адресацію сучасні ОС вважають застарілим режимом роботи і в дійсності обходять її, установлюючи всі сегменти до базової адреси нуля з розміром сегмента 4 *GB*.

- **Архітектура має два основних режими роботи** (див. рис. 1.9)**:**

▪ **Довгий режим** (***Long Mode***). Це основний режим роботи, для реалізації котрого була призначена архітектура. Він уявляє собою комбінацію рідного 64-розрядного режиму роботи процесора і режиму сумісності (*Compatibility Mode*). Режим сумісності дозволяє більшості застарілих 16-розрядних і 32-розрядних додатків запускатися без перекомпіляції в 64-розрядній ОС. Для його використання необхідна 64-розрядна ОС. Програми, котрі використовують реальний або віртуальний режими роботи попередніх x86-процесорів не можуть виконуватися під керуванням 64-розрядної ОС, хіба що тільки з використанням спеціальної програмної емуляції.

▪ **В реальному режимі роботи (*Real Mode*)** використовувалась сегментна адресація пам'яті (адреса комірки пам'яті формувалася з двох чисел: здвигнутої на 4 біта адреси начала сегмента і зміщення комірки від початка сегмента; любому процесу була доступна вся пам'ять комп'ютера). Спершу режим не мав

назви, був названим «реальним» тільки після створення процесорів 80286, підтримуючих режим, що отримав назву **«захищений» (*Protected Mode*)** (режим названо «захищеним», тому що він створювався для «захисту» процесів один від одного — для того, щоб не дозволити процесам мати доступ до не своїх областей пам'яті. Але для процесорів 80286 захищений режим не був по справжньому «захищеним», тому що ці процесори не підтримували сторінкову адресацію пам'яті, вперше реалізовану в процесорах 80386).

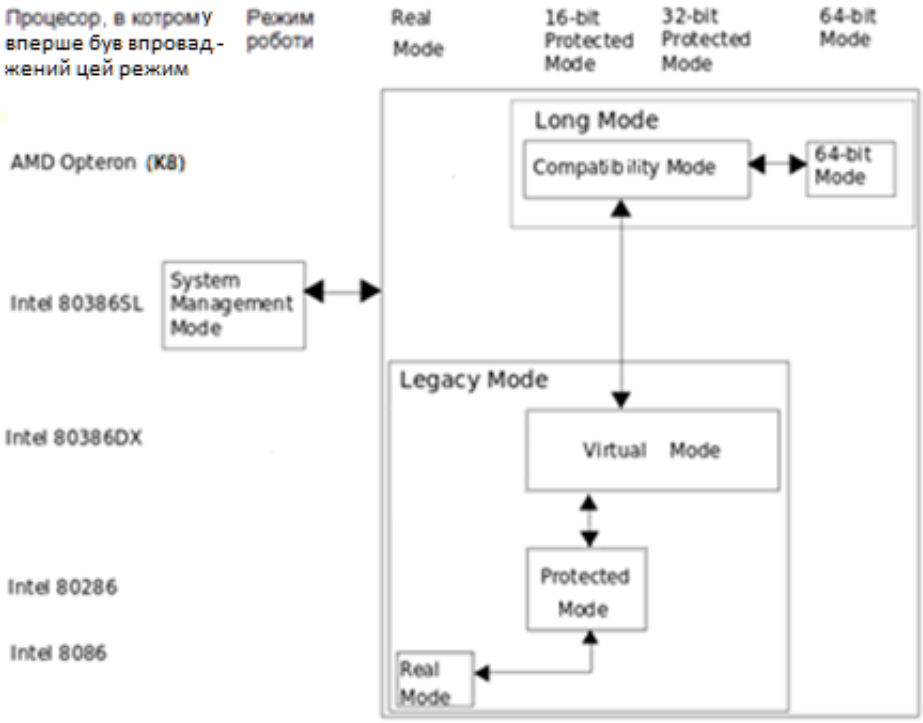

Рисунок 1.9 – Режими роботи, що підтримуються архітектурою *x86-64*

- ***Успадкований режим (Legacy Mode***). В цьому режимі процесор *x86-64* працює під керуванням 32-разрядної ОС. Його робота в цьому режимі подібна роботі процесора *x86* і він може виконувати тільки 16-розрядний и 32-розрядний код. В успадкованому режимі віртуальний адресний простір обмежено 4 *GB*.

- ***Видалення застарілих особливостей*:** Багато особливостей "системного програмування" архітектури *x86* не використовуються в сучасних ОС і тому вони не доступні в довгому режимі. Ці особливості включають: **сегментну адресацію**, хоча *FS* и *GS* сегменти збережені в остаточній формі для використання як додаткові покажчики бази до структур ОС; **механізм переключення контексту** (процес припинення виконання процесором одного потоку із зберіганням всієї

необхідної інформації і стану, необхідних для наступного продовження з перерваного місця) і **віртуальний режим** також не використовується в довгому режимі. Однак, ці особливості повністю підтримуються в успадкованому режимі, таким чином, дозволяючи процесорам *x86-64* виконувати 32-розрядні та 16-розрядні ОС без модифікації. Багато команд в 64-бітовому режимі не підтримуються, наприклад: *збереження/відновлення сегментних регістрів в стек, збереження/відновлення всіх регістрів загального призначення в стек*, *десяткова арифметика*, команди *BOUND* та *INTO*, *"дальні" переходи* та *виклики з безпосередніми операндами*.

- ***Куча (***нерозподілена пам'ять*, heap)* – в інформатиці і програмуванні регіон зарезервованого адресного простору в віртуальній пам'яті додатка, умовна назва структури даних, поверх котрої реалізована динамічна пам'ять процесу.

Куча використовує пам'ять, виділену статично або запрошену динамічно у ОС. Ця пам'ять використовується для розміщення об'єктів, динамічно створених в процесі виконання додатка.

В любий момент часу існування кучі вся пам'ять, на котрій працює куча, розділена на заняту і вільну. Занята пам'ять використана під розміщення об'єктів, уже створених і ще не звільнених до цього моменту часу. Із обсягу вільної пам'яті примітиви роботи з кучею можуть виділяти пам'ять під нові об'єкти.

- ***Програмний стек*** (***stack***) – в інформатиці і програмуванні регіон зарезервованого адресного простору в віртуальній пам'яті додатка для організації структури даних потоку додатка у вигляді списку елементів, організованих по принципу *LIFO (last in — first out,* «останнім прийшов — першим вийшов» - див. рис. 1.10).

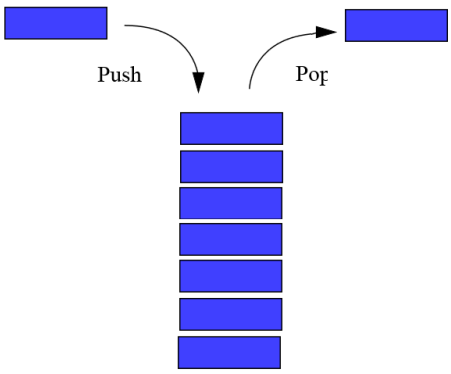

Рисунок 1.10 – Робота з програмним стеком потоку за допомогою команд *Push і Pop*

Для кожного потоку команд (*kernel thread*) ОС виділяє свій *LIFO*-стек з адресного простору додатка. В *LIFO*-стеку зберігається інформація для повернення управління в рамках *hardware thread* із методів (процедур, функцій, підпрограм, обробника переривань і т. п.).

Програмний стек потоку може використовуватися для *різних* потреб, але основне його призначення – відслідковувати місце, куди кожен із викликаних методів повинен повернути управління після свого завершення. Для цього при виклику метода в стек заноситься адреса команди, наступної за командою виклику («адреса повертання»). При наступному вкладеному або рекурсивному виклику, в стек заноситься чергова адреса повертання і т. д. По завершенні викликаного метода виконується команда повертання для переходу по адресі із стека.

В архітектурі *x86-64* в якості вершини стека використовується *Stack Pointer* регістр. При виклику метода **перші шість цілочислових аргументів/покажчиків розміщуються в архітектурних регістрах: *RDI, RSI, RDX, RCX, R8* і *R9*. Тільки починаючи з сьомого аргументу/покажчика (адреса повернення) вони розміщуються в стеку.** Всі спекулятивні *hardware threads*, що породжуються в рамках *kernel thread* (спекулятивні обчислення, ми їх розглянемо у наступному розділі) використовують один і той же стек, але різні фізичні регістри для представлення *Stack Pointer* цього *kernel thread* і перших шести аргументів/покажчиків.

2 СУПЕРСКАЛЯРНА АРХІТЕКТУРА (*CISC-RISC* АРХІТЕКТУРА)

2.1 Призначення *Front-end* суперскалярного ядра

Інструкції змінної довжини роблять їх декодування досить складним. Інструкції *x86-64* із префіксами мають розмір від 1 до 18 байт [1]. Як і в інших архітектурах, потік інструкцій часто переривається потоком керування, таким як умовні розгалуження, переходи, виклики та повернення, які потенційно перенаправляють вибірку інструкцій і створюють непродуктивні бульбашки в конвеєрі інструкцій (див. рис. 2.1, де показана тільки початкова частина цього конвеєра, що отримала назву *Front-end*).

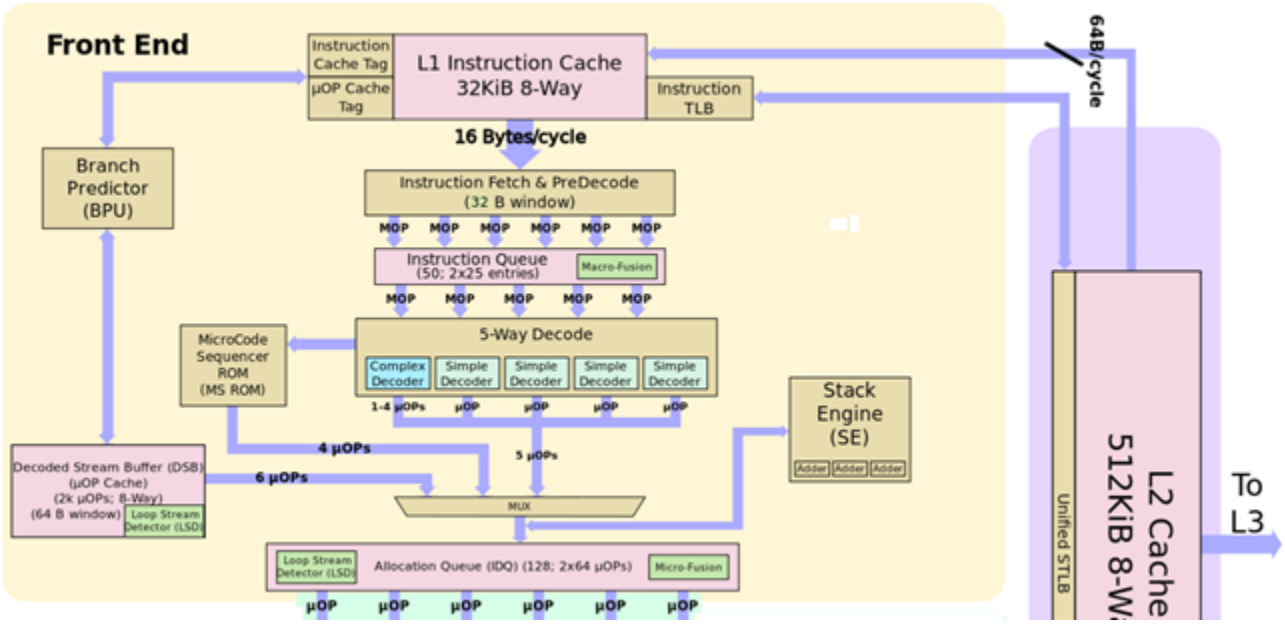

Рисунок 2.1 – *Front-end* ядра *Sunny Cove*

2.1.1 Декодування *CISC* на *RISC*

Передбачені наперед адреси виконуваних *CISC* команд (поперемінно для кожного із 2 потоків — при включеній технології *Hyper-Threading*) видаються для перевірки їх наявності в кеші команд *L1I* та *µOP-Cache* (кеші *RISC* – операцій). Команди *CISC* переводяться в *RISC* операції в декодерах, причому кожна проста команда транслюється в 1 *RISC*, а складна декодується в декілька. В процесі декодування *CISC* на *RISC* використовуються ***мікрозлиття*** та ***макрозлиття***.

Мікрозлиття (Micro-Fusion) – можливість однією *fused-RISC* (*fused-μOPs* в термінології *Intel*) закодувати більше одної *RISC* операції однієї *CISC* команди, щоб зменшити нагрузку на конвеєр для деяких відносно складних команд. Частіше за все так кодується одна обчислювальна операція і один зв'язаний з нею доступ до пам'яті, котрі поділяються на дві окремі *RISC* перед їх плануванням на виконання.

В *Intel* розмір порції одночасно декодуємих *CISC*, що зчитуються з кешу *L1I*, зазвичай складає 16 байтів. Такий підхід є перепоною для векторного коду (*SIMD*), середній розмір команд котрого перевищує 4 байти, тому що в 16 байт не завжди вміщуються навіть 4 *CISC*. *AMD* вирішила цю проблему ще в архітектурі *K10*, розширив порцію декодуємих *CISC* команд до 32 байт, а *Intel* – шляхом введення в схему *Front-end* процесорного ядра двох буферів попереднього декодера (*Instruction Fetch*).

Попередній декодер-довжиномір. З *L1I* чергові 16 байт команд попадають в *Instruction Fetch* свого *hardware thread*, а відтіля, вже чергові 32 байти команд в попередній декодер-довжиномір (*PreDecode*). Попередній декодер-довжиномір знаходить і позначає межі інструкцій, декодує будь-які префікси та перевіряє певні властивості (наприклад, розгалуження). **Пропускна здатність *PreDecode* обмежена 6 *CISC* за цикл**. Таким чином, *PreDecode* відміряє чергові 5 *CISC*, або 6 *CISC*, якщо він розпізнав можливість подальшого макрозлиття двох з них. Біти, що залишилися від обробленої 32 байтної послідовності команд залишаться в *Instruction Fetch* до наступного циклу обробки команд цього *hardware thread*.

Зразу після команди переходу обробка починається з команди по цільовій адресі переходу, інакше – з того байту, перед котрим попередній декодер зупинився тактом раніше при обробці цього потоку команд. Коли становиться відомою наступна адреса, *Sunny Cove* одночасно перевіряє на влучання як кеш *RISC*, так і кеш інструкцій *L1*. Кеш інструкцій *L1* з ємністю 32 КБ і 64 Б рядками (блоками даних) має асоціативність збільшену до 8, що означає, що він віртуально індексований і фізично позначений [9].

Черги декодера. Розмічені команди попадають в одну з двох черг декодера (*Instruction Queue*), по черзі на кожний із одночасно оброблюваних потоків (*hardware threads*), на 25 розмічених команд кожна. По мірі просування розмічених команд в *Instruction Queues* деякі пари сусідніх *CISC* команд, попередньо виявлених *PreDecode*, зливаються в одну *CISC* команду. Цей процес називається ***макрозлиттям (Macro-Fusion)***.

Макрозлиття, в деяких випадках, поєднує арифметичну чи логічну *CISC* інструкцію з заступною *CISC* інструкцією умовного переходу в одну *CISC* інструкцію обчислення і переходу. Такий процес дозволяє економити полосу пропускання конвеєрного ядра на всіх етапах: від декодування *CISC* інструкцій до їх завершення.

Декодування інструкцій. Декодери (*5-Way Decode*) читають *CISC* команди з *Instruction Queues* (по парним тактам з однієї черги, по непарним – з іншої) і переводять їх в *RISC* операції.

Черга інструкцій може надсилати 5 попередньо декодованих інструкцій (або більше за допомогою *Macro-Fusion* на порівняння та розгалуження) до декодерів *x86-64*. Декодери зчитують інструкції *x86-64* і видають регулярні *RISC* фіксованої довжини, які нативно обробляються основним обладнанням. Один з декодерів може обробляти складні інструкції, які видають до 4 *RISC*, а інші 4 декодери обробляють простіші інструкції. Для мікрокодованих інструкцій (тобто >4 *RISC*) мікрокод декодуватиме 4 *RISC*/цикл, доки інструкція не буде завершена, хоча це блокує звичайне декодування. Декодери можуть видавати щонайбільше 5 RISC за цикл – незалежно від того, яке поєднання інструкцій декодується.

Більшість 256-розрядних інструкцій *AVX* обробляються декодерами як прості інструкції завдяки деякій досить розумній роботі над реалізацією в блоках виконання та конвеєрі пам’яті [9]. Усі декодери підтримують мікрозлиття *RISC*, щоб інструкції між регістром і пам’яттю можна було декодувати як одну *RISC*. Спеціальний засіб відстеження вказівника стека також присутній у *Sunny Cove*

(*Stack Engine* на рис. 2.1) і перейменовує вказівник стека, усуваючи послідовні залежності та видаляючи ряд *RISC*.

2.1.2 *Stack Engine* – пристрій обробки команд роботи із стеком

CISC x86-64 має спеціальні команди для роботи із стеком. Такі інструкції, як *PUSH, POP,* а також *CALL* і *RET*, працюють на вказівнику стека (*RSP*). Без будь-якого спеціалізованого апаратного забезпечення такі операції потрібно було б надсилати на *Back-end* частину ядра для виконання за допомогою *ALU* загального призначення, використовуючи частину пропускної здатності та використовуючи ресурси планувальника та блоків виконання. Починаючи з *Pentium M, Intel* використовує для цієї цілі спеціальний пристрій – *Stack Engine*. В ядрі *Sunny Cove Stack Engine* має набір із трьох спеціальних суматорів, які використовуються для виконання й усунення *RISC* для оновлення стека (тобто можуть оброблятися до трьох *RISC* роботи із стеком за цикл). *Stack Engine* знаходиться після декодерів і контролює потік *RISC*, що проходить повз нього. *RISC* операції, що зв'язані з модифікацією стека, виконуються в *Stack Engine*, що зменшує навантаження на наступні сегменти конвеєра команд, що знаходяться в *Back-end* частині ядра.

2.1.3 Кеш інструкцій *L0*

***RISC*-кеш** *(µOP-Cache* в термінології *Intel),* який вперше був представлений в архітектурі *Sandy Bridge*, діє як кеш інструкцій *L0*, та містить декодовані *RISC* фіксованої довжини. Кеш *RISC* віртуально адресований і включений у кеш інструкцій *L1*. Після того, як інструкції декодовано, *RISC* надсилаються до *RISC*-кешу та до *Back-end* частини ядра для розподілу, перейменування та виконання поза порядком. Подальші вибірки інструкцій по тій самій адресі повинні потрапляти в кеш *RISC*, оминаючи звичайну вибірку інструкцій і їх декодування.

Використання *RISC*-кешу покращує продуктивність і енергоспоживання. Кеш *RISC* концептуально є підмножиною кешу інструкцій *L1*, який кешується в декодованій формі. Основні цілі полягають у покращенні пропускної здатності

Front-end ядра процесора та усуненні декодування з критичного шляху в конвеєрі команд.

У *Sunny Cove* потік інструкцій розділений на ряд вікон . Кожне вікно – це 32Б інструкцій, які, можливо, обмежені будь-якими прийнятими гілками у вікні [9]. Відображення між кешем інструкцій і кешем *RISC* відбувається з деталізацією повного вікна 32Б. Після того, як усе вікно було декодовано та надіслано до *Back-end*, воно також вставляється в кеш *RISC*. Важливо те, що цей процес збирання відбувається паралельно з нормальною роботою *Front-end* та не викликає жодних затримок [9].

Звернення до *RISC*-кешу має кілька переваг, зокрема зменшення довжини конвеєра шляхом усунення енергоємних етапів декодування інструкцій і забезпечення ефективної пропускної здатності 32Б інструкцій за цикл. Для новіших інструкцій *SIMD* обмеження вибірки 16Б було проблематичним, тому кеш *RISC* чудово поєднується з такими розширеннями, як *AVX* [9].

Рядки в кеші *RISC* адресуються за адресую першої декодованої інструкції *x86-64*, яка зберігається в рядку, тому кеш адресується віртуально. Записи позначаються тегами, щоб два потоки могли статично розділити кеш *RISC* [9].

Кеш *RISC* організований у 48 рядків з асоціативністю 8, із 6 *RISC* на рядок. Це визначає його загальну ємність в 2,25 КБ *RISC*. Кеш *RISC* строго включений у кеш інструкцій *L1*. Кожен рядок також містить метадані, включаючи кількість дійсних *RISC* у рядку та довжину інструкцій *x86-64*, що відповідають рядку кешу *RISC*. Кожне вікно розміром 32Б, відображене в *RISC*-кеші, може охоплювати 3 із 8 шляхів у наборі, максимум 18 *RISC* – приблизно 1,8Б/*RISC*. **Якщо вікно 32Б має більше 18 *RISC*, воно не може помістится в кеш *RISC* і має використовувати традиційний *Front-end*. Мікрокодовані інструкції не зберігаються в *RISC*-кеші**, а натомість представлені вказівником на мікрокод в *ROM* [9].

Згідно з даними *Intel, RISC*-кеш працює як кеш інструкцій *L1* розміром 9 КБ і 80% всіх звернень до *L1I* відбуваються як звернення до *RISC*-кеш.

Повертаючись трохи назад до вибірки інструкцій, передбачений адрес вікна інструкцій використовується для дослідження тегів в *RISC*-кеші паралельно зі звичайним зверненням до *L1I*. Влучання в теги кешу *RISC* дає ідентифікатори рядку та шляху, які поміщаються в чергу відповідності, а потім використовуються для доступу до масивів даних кешу *RISC*. Коли відбувається звернення, з *RISC*-кешу у відповідну чергу розподілу або буфер (*Allocation Queue* на рис. 2.1) передаються всі *RISC* у вікні інструкцій розміром 32 Б (тобто до 3 рядків). Цілком імовірно, що масиви даних кешу *RISC* зчитують лише один рядок кожного циклу, тому для завершення звернення може знадобитися кілька циклів [9]. Кеш *RISC* може надсилати до 6 *RISC* за цикл до *Allocation Queue*. Однак звернення до кешу *RISC* може охоплювати повне вікно інструкцій розміром 32Б. Це подвоює пропускну здатність традиційного *Front-end*, який обмежений 16Б вибірками інструкцій з *L1I*. Додаткова пропускна здатність особливо корисна, якщо середня довжина інструкції перевищує 4 байти; наприклад, інструкції *AVX* використовують 2- або 3-байтовий префікс.

Влучання в *RISC*-кеш повністю обходить вибірку інструкцій і апаратне декодування, заощаджуючи енергію та покращуючи постачання *RISC* від *Front-end*. При цьому, більша частина блоків *Front-end*, що знаходяться вище кешу *RISC* (див. рис.2.1), вимикаються і тим самим зменшується кількість теплоти, що виділяється ядром, а частота роботи блоків *Front-end*, що знаходяться далі по конвеєру, підвищується до досягнення гранично допустимої температури. Після влучання кеш *RISC* обчислює адрес наступного вікна інструкцій, використовуючи поля довжини інструкції, які зберігаються в кожному рядку кешу *RISC* [9].

У разі промаху в *RISC*-кеш вибірка і декодування інструкцій продовжуються, як описано раніше із *L1I*. При цьому на деяких тактах роботи *RISC*-кешу може зніматися синхронізація для зберігання електроенергії [9].

Політики розміщення рядків кешу *RISC* і політики вигнання були розроблені таким чином, щоб кожне вікно розміром 32Б було повністю кешованим (або не було кешованим взагалі) [9].

Кеш-пам’ять *RISC* є одним з найбільш перспективних обладнань у *Sunny Cove*, оскільки вона одночасно зменшує енергоспоживання та покращує продуктивність. Це дозволяє уникнути енергоємного декодування команд *x86-64*, яке охоплює кілька етапів конвеєра та вимагає досить дорогого апаратного забезпечення для обробки нестандартного набору інструкцій [9].

2.1.4 Черга розподілу (*Allocation Queue)* та вбудований в неї кеш для малих циклів (*Loop Stream Detector*)

Усі *RISC* з традиційного *Front-end* та кешу *RISC* зрештою доставляються в *Loop Stream Detector (LSD),* який діє як кеш для малих циклів і фактично може виключити з роботи як кеш інструкцій *L1*, так і кеш *RISC*. При цьому, більша частина блоків *Front-end*, що знаходяться вище *LSD,* відключаються і відбуваються дії, що раніше описані для *RISC*-кешу [9].

У *Sandy Bridge* чергу *RISC* до *Allocator* (*Allocation Queue*) із 28 записів було відтворено для кожного потоку. Однак у *Ivy Bridge* і *Haswell* (56 записів), у *Skylake* і *Sunny Cove* (128 записів) чергу *RISC* було об’єднано в структуру з одним записом, яка статично розділена, коли активні два потоки. Відмінність полягає в тому, що коли виконується один потік, усі 128 записів у *Allocation Queue* (у *Sunny Cove*) доступні для кешування циклу та постановки в чергу, що дозволяє краще використовувати доступні ресурси [9]. В *Allocation Queue* реалізуються також розглянуті нами раніше *Micro Fusion.*

В цілому, робота *Front-end* полягає в тому, щоб постійно доставляти достатню кількість *RISC* з потоку інструкцій до *Back-end*, щоб він був, бажано, постійно зайнятим. Навіть найпродуктивніший ЦП із позачерговим виконанням забезпечить погані результати без потужного *Front-end*. Для будь-якого сучасного ЦП з архітектурою *x86-64* це досить складно реалізувати, бо доставка потоку інструкцій часто переривається розгалуженнями, і прийняте розгалуження може створити бульбашку в конвеєрі, оскільки вибірка інструкцій перенаправляється на нову адресу. Декодування байтів інструкцій *x86-64* у *RISC*

ускладнюється природою змінної довжини інструкцій *x86-64*, безліччю префіксів і надзвичайно складними мікрокодованими інструкціями [1]. Архітектори доклали величезних зусиль, щоб удосконалити всі ці аспекти *Front-end*. *Allocation Queue* може приховати бульбашки конвеєра, створені прийнятими гілками, і здійснювати ефективне «зшиття між прийнятими гілками».

2.2 Принципи побудови кеш-пам’яті

Рядки кеш-пам’яті тимчасово дублюють деяку кількість комірок основної пам’яті *RAM*, що знаходяться в *RAM* поряд і носять назву блока даних, для пришвидшення доступу до них. Таким чином, кеш-пам’ять – це апаратне забезпечення, яке зменшує час доступу до даних у пам’яті, зберігаючи частину часто використовуваних даних основної пам’яті. Вона є меншою та швидшою, ніж основна пам'ять, та має декілька рівнів ієрархії: рівні *L1* та *L2* містяться у кожному ядрі процесора; частина кешу *L3* міститься в кожному ядрі – разом вони утворюють кеш *L3*, що є загальним для всіх ядер; старші моделі процесорів, що підтримують технологію *Intel vPro,* мають також пам’ять *Side (eDRAM)*, що є вбудованою в північний міст, який, починаючи з мікроархітектури *Nehalem* (див. табл. 1.1) знаходиться на кристалі мікропрооцесора (цю пам’ять можна розглядати, як кеш рівня *L4,* що знаходився раніше на материнській платі). Для пришвидшення доступу до пам’яті кеш *L1* реалізується як асоціативна пам’ять, а для забезпечення паралельної роботи різних сегментів в конвеєрі команд процесорного ядра *L1* розбивається додатково, у відповідності з принципами Гарвардської архітектури [1], на *L1I* та *L1D* (відповідно, кеш-пам’ять *L1* для представлення команд та даних). Кеші *L1D* всіх ядер об’єднані між собою апаратним механізмом когерентності кешів.

2.2.1 Структура рядка кеш-пам’яті

Рядок кешу зазвичай має таку структуру:

Рисунок 2.2 – Структура рядка кеш-пам’яті

Блок даних (*data block*) містить фактичні дані, що зберігаються в цьому рядку кешу. Тег (*tag*) містить старшу частину фізичного адресу даних в основній пам'яті. "Розмір" кешу - це обсяг даних основної пам'яті, який він може вмістити. Цей розмір можна розрахувати як кількість байтів, що зберігаються в кожному блоці даних, помножену на кількість блоків, що зберігаються в кеші. Біти тегу, прапора (*flag bits*) та коду виправлення помилок (якщо цей код використовується) не входять до розміру, хоча вони впливають на фізичну область кешу.

Flag bits. Для рядку кешу інструкцій потрібен лише один біт прапора: ***valid*** («дійсний») біт. Цей біт вказує, чи було завантажено в цей рядок кешу блок даних з дійсними командами. Під час увімкнення апаратне забезпечення встановлює всі дійсні біти в усіх кешах на "недійсні" (*"invalid"*). В рядках кешів *L1D* значення цього біту також може встановлюватися на «недійсний» в інший час, наприклад, коли апаратне забезпечення відстеження механізму кеш-когерентності у кеші одного ядра чує широкомовну адресу від іншого ядра та розуміє, що певний блок даних у локальному кеші зараз є застарілим і має бути позначений як «недійсний».

Для кешу даних зазвичай потрібні три біти прапора на рядок кешу: *valid* біт; ***dirty*** («брудний»); назва третього біта прапора залежить від використовуваного механізму кеш-когерентності (ми це розглянемо пізніше). Наявність «брудного» біта вказує на те, що відповідний рядок кешу було змінено після його створення з відповідним *tag* та значеннями бітів прапора: «дійсний» і «небрудний», тобто ядро процесора записало дані в цей рядок і нове значення ще не поширилося до іншого кешу.

2.2.2 Структура адреси блоку даних, що заходиться в рядку кешу

Фізична адреса блоку даних в *RAM*, що знаходиться в рядку кешу, розділена на (див. рис. 2.3): *tag (*тег), *index (*індекс) і *block offset* (зміщення блоку).

Рисунок 2.3 – Структура адреси блоку даних, що заходиться в кеш-рядку

Індекс описує, у який рядок кешу було поміщено дані. Довжина індексу становить *log2 r* бітів для *r* рядків кешу.

Поле *tag* містить старшу частину фізичної адреси блоку даних в *RAM*, що знаходиться в рядку кешу. Було б дуже нераціональним наділяти кожен байт в кеш-пам'яті адресним полем, що вказувало б на його місцезнаходження в *RAM*, тому що це привело б до дуже великих апаратурних втрат. Крім того, така кеш-пам'ять була б дуже поганою з точки зору продуктивності. Тому, значно зручнішим є адресувати деякі групи сусідніх байтів, котрі й будуть формувати блок даних строки кешу. Широко використовуються блоки даних по 32, або 64 байти. Їх розмір може досягати навіть 1024 байтів. Звичайно, що його розмір повинен бути еквівалентним двійці в якійсь цілій додатній ступені. Однак, навіть якщо вимога по розміру виконана, не кожна група сусідніх байт може бути кешована по причині додаткового обмеження, відомого як ***адресне вирівняння (address alignment)***. Іншими словами, група сусідніх байтів може бути поміщеною в строку кеш-пам'яті тільки тоді, коли її початкова адреса вирівняна по границі, що дорівнює розміру інформаційного блоку. Наприклад, 64-байтний блок може бути заповнений інформацією із оперативною пам'яті, що знаходиться за шістнадцятковими (десятковими) адресами: *00-3F (00-63), 64-7F (64-127)* і т. д. Крім інших переваг, це просте правило дозволяє скоротити число адресних байтів в розрахунку на одну строчку кешу. Якщо точніше, то на 6 в вищенаведеному прикладі: $log_2(64) = 6$. Блок даних може бути або повністю

заповненим дійсною інформацією, або повністю порожнім, що рівносильно бути заповненим недійсною інформацією.

Кожна строчка кеш-пам’яті зазвичай має якусь надлишкову інформацію для здійснення контроля за помилками. Поля даних захищаються або перевіркою парності (*parity checking*) на рівні окремих байтів, або, значно ріже, одним із протоколів перевірки і корекції помилок (*ECC — Error Checking and Correcting*) на рівні всієї строки, більшість із котрих ґрунтуються на алгоритмі Хеммінга (*the Hamming algorithm*).

2.2.3 *TLB* буфери (*Translation Look-aside Buffers*) та віртуальна пам’ять

В обчислювальній техніці віртуальна пам’ять (також віртуальне сховище) – це техніка управління пам’яттю, яка забезпечує ідеалізовану абстракцію ресурсів зберігання, які фактично доступні на даній машині, що створює у користувачів ілюзію дуже великої основної пам’яті. Переваги віртуальної пам’яті включають звільнення програм від необхідності керувати спільним простором пам’яті, підвищену безпеку завдяки ізоляції пам’яті та можливість концептуально використовувати більше пам’яті, ніж може бути фізично доступним, за допомогою техніки підкачки.

Віртуальна пам'ять полегшує програмування додатків, приховуючи фрагментацію фізичної пам'яті; шляхом делегування ядру ОС тягаря керування ієрархією пам’яті (усуваючи потребу програмі явно обробляти накладення); і, коли кожен процес виконується у своєму власному виділеному адресному просторі, шляхом уникнення необхідності переміщення програмного коду або доступу до пам’яті за допомогою відносної адресації. Таблиця сторінок – це структура даних, яка використовується системою віртуальної пам’яті для збереження відображення між віртуальними та фізичними адресами. Віртуальні адреси використовуються процесом доступу, тоді як фізичні адреси використовуються апаратним забезпеченням, або, точніше, підсистемою оперативної пам’яті. В операційних системах, які використовують віртуальну пам'ять, для кожного процесу створюється враження, що він працює з великими

безперервними ділянками пам'яті. Фізично пам’ять кожного процесу може бути розподілена по різних областях фізичної пам’яті або може бути переміщена (вивантажена) в інше сховище, як правило, на жорсткий диск. Коли процес запитує доступ до даних у своїй пам’яті, система віртуальної пам’яті несе відповідальність за зіставлення віртуальної адреси, наданої процесом, із фізичною адресою фактичної пам’яті, де ці дані зберігаються. Таблиця сторінок– це місце, де система віртуальної пам’яті зберігає свої відображення віртуальних адрес у фізичні адреси, при цьому кожне відображення також відоме як вхід таблиці сторінок. Апаратне забезпечення ядра, автоматично перетворює віртуальні адреси на фізичні. Програмне забезпечення всередині операційної системи може розширити ці можливості, щоб забезпечити віртуальний адресний простір, який може перевищувати обсяг реальної пам’яті, і, отже, використовувати більше пам’яті, ніж фізично присутнє в комп’ютері.

Апаратне забезпечення ядра, котре називається *memory management unit (MMU)* разом з ОС *TLB miss handler* (обробником промаху в *TLB*) зберігають в спеціальному кеші деякі відображення із таблиці сторінок ОС. Їх робота полягає в тому, щоб наперед підвантажувати в цей спеціальний кеш відображення із таблиці сторінок ОС, які зможуть знадобитися в найближчих обчисленнях. Цей спеціальний кеш носить назву *Translation Look-aside Buffers (TLB)* і є асоціативним кешем.

При трансляції віртуальної адреси в фізичну в ядрі процесора, спершу відбувається пошук в *TLB* за віртуальною адресою. Якщо відбулося влучання в *TLB (TLB hit),* з рядку *TLB* видається фізична адреса, за якою і відбувається пошук потрібного блоку даних у відповідному кеші команд чи даних.

Якщо відбувся промах в *TLB (TLB miss),* ОС *TLB miss handler* буде шукати потрібну віртуальну адресу в таблиці сторінок, щоб знайти відповідну фізичну адресу (здійснює обхід сторінок - *a page walk*). Якщо відображення існує, то воно записується у відповідний рядок *TLB* (див. рис. 2.4).

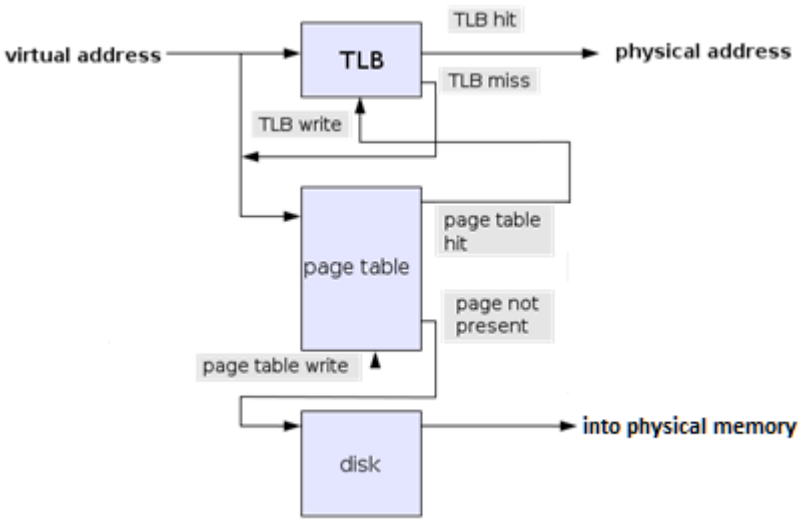

Рисунок 2.4 – Процес трансляції віртуальної адреси в фізичну

Пошук в таблиці сторінок може не вдатися з двох причин:

• Пошук може завершитися помилкою, якщо для віртуальної адреси немає перекладу, тобто віртуальна адреса недійсна. Зазвичай це відбувається через помилку програмування і операційна система повинна вжити певних заходів, щоб вирішити цю проблему.

• Пошук також завершується помилкою, якщо сторінка наразі не знаходиться у фізичній пам'яті. Це станеться, якщо запитану сторінку було переміщено з фізичної пам’яті, щоб звільнити місце для іншої сторінки. У цьому випадку сторінка передається до додаткового сховища, розташованого на носії, наприклад, на жорсткому диску (це вторинне сховище часто називають «розділом підкачки», якщо це розділ диска або «файл підкачки» («файл сторінки»), якщо це файл). У цьому випадку ОС повертає потрібну сторінку з диску у фізичну пам’ять.

Коли фізична пам'ять не заповнена, сторінка записується у фізичну пам'ять, при цьому таблиця сторінок і *TLB* оновлюються.

Коли фізична пам’ять заповнена, якась із сторінок у фізичній пам’яті вивантажується на диск, щоб звільнити місце для потрібної сторінки. При цьому таблиця сторінок і *TLB* також оновлюються.

2.2.4 Вміст таблиці сторінок

Таблиця сторінок містить відображення між віртуальною адресою сторінок процесів та адресою фізичної пам'яті. Вона також містить допоміжну інформацію про сторінку, таку як: біт присутності (*present bit*) сторінки в фізичній пам'яті; брудний або модифікований біт (*dirty bit*) – показує, що сторінка в фізичній пам'яті була модифікована і при її видаленні з фізичної пам'яті вона повинна бути збереженою на диску; ідентифікаційний номер (*ID)* процесу – містить інформацію для системи керування віртуальною пам'яттю комп'ютера, щоб вона знала, які сторінки з яким процесом асоціювати; *No-Execute* або *XD*-біт (див. підрозділ 1.6). Пов'язування *ID* зі сторінками віртуальної пам'яті комп'ютера допомагає ОС у виборі сторінок процесів для виведення на таблицю сторінок, оскільки сторінки, пов'язані з неактивними процесами, зокрема процеси, основна кодова сторінка яких була вивантажена, менш імовірно знадобляться негайно, ніж сторінки, що належать активним процесам.

Таким чином, **таблиця сторінок містить відображення між віртуальною адресою сторінок множини існуючих процесів, що носять назву тіньових таблиць сторінок та адресою фізичної пам'яті комп'ютера.**

2.2.5 Трансляція адрес другого рівня (*SLAT*)

Трансляція адрес другого рівня (*Second Level Address Translation - SLAT*) – це програмна технологія віртуалізації з підтримкою спеціального апаратного забезпечення, яка дає змогу уникнути накладних витрат, пов'язаних із керуванням тіньовими таблицями сторінок (*nested paging*).

AMD підтримує *SLAT* через технологію швидкого індексування віртуалізації (*Rapid Virtualization Indexing - RVI*) з моменту представлення своїх процесорів *Opteron* третього покоління (кодова назва *Barcelona*). Реалізація *SLAT* від *Intel*, відома як *Extended Page Table (EPT)*. Вона була впроваджена, починаючи з мікроархітектури *Nehalem*.

Сучасні мікропроцесори використовують поняття фізичної пам'яті та віртуальної пам'яті. Запущені процеси використовують віртуальні адреси. Коли інструкція запитує доступ до пам'яті, ядро мікропроцесора перетворює віртуальну адресу у фізичну адресу за допомогою *TLB*, що містить інформацію із таблиці сторінок. **Під час роботи процесу йому виділяється віртуальна пам’ять головної системи, яка служить фізичною пам’яттю для гостьової системи.** Процес трансляції адреси відбувається також і у гостьовій системі. Це збільшує вартість доступу до пам’яті, оскільки трансляцію адреси потрібно виконувати двічі – один раз у гостьовій системі (за допомогою програмної емуляції тіньової таблиці сторінок) і один раз у системі хоста (за допомогою апаратної таблиці сторінок).

Щоб зробити цей переклад більш ефективним, постачальники процесорів реалізували технології, які зазвичай називають *SLAT*. Враховуючи кожну гостьову фізичну адресу як хост-віртуальну адресу, невелике розширення апаратного забезпечення, що використовується для проходження невіртуалізованої таблиці сторінок, може проходити таблицю сторінок хоста. За допомогою багаторівневих таблиць сторінок таблицю сторінок хоста концептуально можна розглядати як вкладену в таблицю сторінок гостьової системи. Апаратний обробник таблиці сторінок може розглядати додатковий рівень перекладу майже як додавання рівнів до таблиці сторінок.

Використовуючи *SLAT* і багаторівневі таблиці сторінок, кількість рівнів, які потрібно пройти, щоб знайти переклад, подвоюється, коли фізична адреса гостьового типу має такий самий розмір, як гостьова віртуальна адреса, і використовуються сторінки того самого розміру. Це підвищує важливість кешування значень із проміжних рівнів таблиць сторінок хоста та гостьових сторінок. Також корисно використовувати великі сторінки в таблицях сторінок хоста, щоб зменшити кількість рівнів (наприклад, у *x86-64* використання сторінок розміром 2 МБ видаляє один рівень у таблиці сторінок). Оскільки пам’ять, як правило, виділяється віртуальним машинам із грубою деталізацією, використання великих сторінок для гостьового фізичного перекладу є

очевидною оптимізацією, що зменшує глибину пошуку та пам’ять, необхідну для зберігання таблиць сторінок хоста.

2.2.6 Плоска модель пам'яті

Плоска модель пам’яті або лінійна модель пам’яті відноситься до парадигми адресації пам’яті, в якій пам’ять бачиться програмою як єдиний безперервний адресний простір [1]. Ядро може напряму (лінійно) звертатися до всіх доступних ділянок пам’яті без необхідності вдаватися до будь-яких схем сегментації пам’яті чи сторінок.

Управління пам’яттю та трансляція адрес можуть бути реалізовані поверх моделі плоскої пам’яті, щоб полегшити: функціональність ОС, захист ресурсів, багатозадачність або збільшити ємність пам’яті за межі, накладені фізичним адресним простором мікропроцесора.

Ключова функція плоскої моделі пам’яті полягає в тому, що весь простір пам’яті є лінійним, послідовним і безперервним.

ЇЇ переваги: простий інтерфейс для програмістів; найбільша гнучкість; мінімальна кількість апаратного забезпечення; максимальна швидкість виконання програми.

ЇЇ недоліки: не підходить для звичайних обчислювальних або багатозадачних ОС, якщо вони не вдосконалені додатковим апаратним/програмним забезпеченням керування пам’яттю. Всі сучасні мікропроцесори мають апаратну підтримку цієї моделі пам’яті. Це дозволяє, наприклад, ОС *Linux* на основі моделі плоскої пам’яті реалізувати розширену технологію керування та захисту пам’яті.

Таким чином, сучасні мікропроцесори використовують також і поняття лінійної пам'яті, поряд із поняттями фізичної та віртуальної пам'яті (див. підрозділ 2.2.5).

2.2.7 Сегментована модель пам'яті в архітектурі *x86*

Ця модель є подібною до моделі сторінкової пам'яті, але сторінковий доступ досягається шляхом неявного додавання двох відносно зміщених регістрів: *сегмента* та *зсуву*. В підсумку отримувалась модель організації пам’яті процесу із змінними межами сторінок, що виявлялась у деяких випадках більш ефективною та гнучкою, ніж модель сторінкової пам'яті. Однак, вона є найбільш складною і незручною з точки зору програміста та реалізації компілятора. В наслідку того, що сторінки могли перекриватися і велика комбінація параметрів віртуальної пам’яті: *сегмента* та *зсуву* перетворювалася в одну і туж фізичну адресу виникали поганий захист та ізоляція ресурсів. Це приводило до збільшення ймовірності програмних помилок.

В сімействі мікропроцесорів *x86*, цю модель було впроваджено, починаючи з *Intel 8086, 8088*. З тих пір ця модель пам’яті зберігалася в наступних мікропроцесор *x86*, які стали використовуватися для забезпечення роботи багатозадачної ОС, але рідко працювали у сумісному сегментованому режимі. У межах архітектури *x86 (x86-64),* під час роботи в реальному (сумісному) режимі, фізична адреса обчислюється як: *Адреса = 16 × сегмент + зсув* (тобто вміст 16-бітного сегментного регістру зсувається вліво на 4 біти та додається до 16-бітового зсуву, що призводить до 20-бітної адреси.)

2.2.8 Модель сторінкової пам'яті

Підходить для загального дизайну багатозадачної ОС та захисту і розподілу ресурсів. На її основі реалізується віртуальна пам'ять. Проте ця модель потребує більше апаратних ресурсів ядра, приводить до деякого зменшення швидкості виконання програми та є більш складною для її розуміння програмістом.

Це модель пам'яті була впроваджена в сімейство мікропроцесорів *x86*, починаючи з *Pentium Pro* і використовувалась для перетворення 32-розрядних віртуальних адрес процесів у 36-розрядні фізичні адреси основної пам’яті мікропроцесора (це називається розширенням фізичної адреси – *Physical Address Extension (PAE)*).

В таблиці 2.1 представлені три режими роботи із сторінковою пам’яттю, котрі підтримуються в сучасних мікропроцесорах фірми *Intel* з архітектурою *x86-64*.

Таблиця 2.1 – Три режими роботи із сторінковою пам’яттю в архітектурі *x86-64* [2].

Режим роботи сторінкової пам'яті	PG біт в регістрі CR0	PAE біт в регістрі CR4	LME біт в регістрі IA32_EFER	Віртуальна адреса [бітів]	Фізична адреса [бітів]	Розмір сторінки	Підтримка Execute-Disable	Підтримка PCIDs і ключів захисту
Немає	0	Недоступний	Недоступний	32	32	Немає	Ні	Ні
32-bit	1	0	0	32	до 40	4 КБ 4 МБ	Ні	Ні
PAE	1	1	0	32	до 52	4 КБ 2 МБ	Так	Ні
4(5)-рівневий	1	1	1	48 (57)	46 (52)	4 КБ 2 МБ 1 ГБ	Так	Так

Режими роботи із сторінковою пам’яттю відрізняються наступними деталями: розміром віртуальних адрес, які можна перекласти; розміром сторінки; підтримкою прав доступу виконання; підтримкою ключів захисту; підтримкою заборони отримувати інструкції зі сторінки.

За допомогою 4(5)-рівневої таблиці сторінок програмне забезпечення може активувати засіб, за допомогою якого логічний процесор кешує інформацію для кількох віртуальних адресних просторів. Ядро процесору може зберігати кешовану інформацію в *TLB*, коли програмне забезпечення перемикається між різними просторами віртуальних адрес. При цьому програмне забезпечення може ввімкнути функцію, за допомогою якої кожна віртуальна адреса пов’язується з ключем захисту. Програмне забезпечення може використовувати новий контрольний регістр, щоб визначити для кожного ключа захисту, як програмне забезпечення може отримати доступ до віртуальних адрес, пов’язаних із цим ключем захисту.

Сучасні процесори *x86-64* використовують 4(5)-рівневу структуру таблиці сторінок під час роботи в 64-розрядному режимі. Подібна ситуація виникла, коли 32-розрядні процесори *IA-32* використовували два рівні, дозволяючи до чотирьох гігабайт пам'яті (як віртуальної, так і фізичної). Щоб підтримувати більше 4 ГБ

оперативної пам’яті, було визначено додатковий режим трансляції адрес під назвою *Physical Address Extension (PAE)*, що включає третій рівень. Це здійснюється шляхом встановлення біта *PAE* в регістрі *CR4* (див. табл. 2.1). Подібним чином нове розширення вмикається встановленням біта 12 регістра *CR4* (відомого як *LA57*). Це використовується лише тоді, коли процесор працює в 64-розрядному режимі, і може бути змінено лише тоді, коли він не працює. Якщо біт не встановлений, процесор працює з 4-рівневою таблицею сторінок. Нові розширення дозволяють до 4 ПБ фізичної пам’яті в порівнянні з максимальним обсягом 64 ТБ на попередніх процесорах. Оскільки додавання п’ятого рівня в таблиці сторінок збільшує віртуальний адресний простір в 512 разів, віртуальний ліміт збільшився з 256 ТБ до 128 ПБ. Додаткові дев'ять бітів віртуальної адреси індексують нову таблицю, тому, хоча раніше використовувалися біти з 0 по 47, тепер використовуються біти з 0 по 56 (див. рис. 2.5).

Рисунок 2.5 – Діаграма 5-рівневої структури таблиці сторінок (регістр *CR3* вказує на початкову адресу кореневої таблиці сторінок в *RAM*).

Якщо *CR0.PG = 0* (див. табл. 2.1), таблиці сторінок не використовуються. При цьому логічний процесор розглядає всі лінійні адреси як фізичні адреси, а біти *CR4.PAE* та *IA32_EFER.LME*[біт 8] ігноруються процесором, як і *IA32_EFER.NXE*[біт 11] (вмикає обмеження доступу до сторінки, запобігаючи вибірці інструкцій зі сторінок *PAE* із встановленим бітом *XD*).

Таблиці сторінок увімкнено, якщо *CR0.PG = 1*. Їх можна ввімкнути, лише якщо захист увімкнено (*CR0.PE = 1*). Якщо таблиці сторінок увімкнено, використовується один із трьох режимів їх роботи. Значення бітів: *CR4.PAE* та *IA32_EFER.LME* визначають, який режим підкачки використовується: якщо *CR0.PG = 1* і *CR4.PAE = 0*, буде працювати режим, що називається 32-bit; якщо *CR0.PG = 1, CR4.PAE = 1* і *IA32_EFER.LME*[біт 8] = *0* – працює режим *PAE*; якщо *CR0.PG = 1, CR4.PAE = 1* і I*A32_EFER.LME*[біт 8] = *1*, використовується 4(5) - рівневий режим.

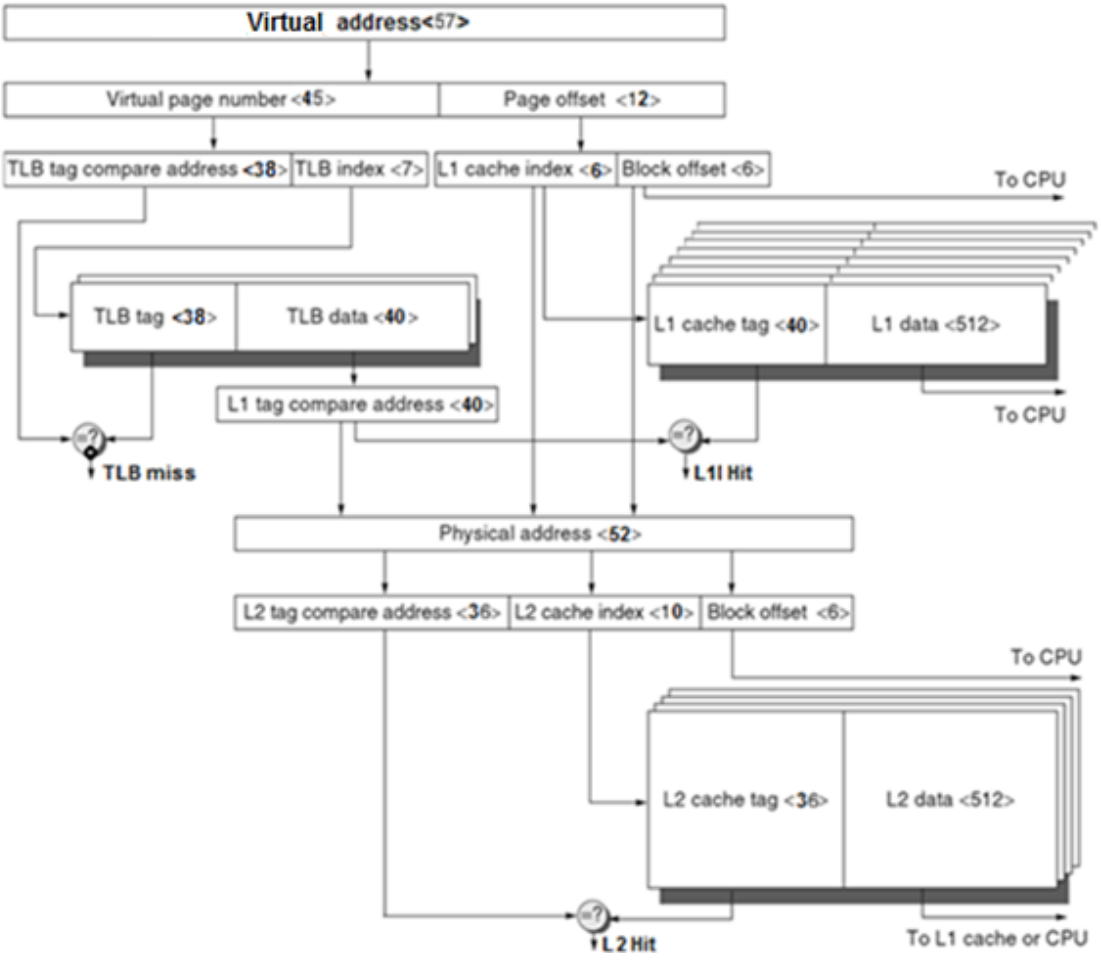

Рисунок 2.6 – Перетворення 57 бітних віртуальних адрес додатка в 52 бітні фізичні адреси *RAM* для кешів *L1I* і *L2* ядра (5-рівнева таблиця сторінок).

Сенс організації кеш пам'яті мікропроцесора полягає в тому, щоб як можливо більша кількість звернень до фізичного простору оперативної пам'яті здійснювалась як звернення до кешів команд і даних першого рівня.

Для організації ефективної підкачки в кеш пам'ять потрібної інформації з оперативної пам'яті, сучасні мікропроцесори містять в своєму складі спеціальну апаратуру. До цієї апаратури відносяться: ***A-box*** – здійснює розрахунки при перетворенні адресів; **буфери *TLB,* що використовуються для пришвидшення трансляції віртуальної адреси процесу в адресу фізичної пам'яті комп'ютера,** а також інша спеціальна процесорна логіка.

По аналогії з звичайними кешами, вони можуть бути розділеними (***I-TLB, D-TLB***) або уніфікованими ***U-TLB.*** Буфер ***I-TLB*** зберігає записи, що відносяться до сторінок з командами. Буфер ***D-TLB*** – до сторінок з даними. Різні *TLB* буфери можуть характеризуватися різними: асоціативністю, кількістю портів читання/запису, політикою заміщення записів і т. п.

На рис. 2.6 показаний приклад роботи блоку *MMU* мікропроцесорного ядра [1] при перетворенні 57 бітних віртуальних адрес додатка з розміром сторінок 4КБ в 52 бітні фізичні адреси оперативної пам'яті для кеша *L1I* з 8-канальною асоціативністю і 64 записами (ємність кеша 32КБ), а також кеша *L2* мікропроцесора з 4-канальною асоціативністю і 1024 записами (ємність кеша 256 КБ). При цьому *TLB* має 2-канальну асоціативність і 128 записів. В обох кешах використовуються 64-байтові блоки даних.

2.2.9 Асоціативність кеш-пам'яті та *TLB* буферів

Одна із фундаментальних характеристик кеш-пам'яті – ***рівень її асоціативності, що відображує її логічну сегментацію.*** Послідовний перебір всіх строк кеша в пошуках потрібних даних потребував би десятків тактів і таким чином втратився б увесь виграш від використання кеша. Тому комірки *RAM* жорстко прив'язуються до строчок кеш-пам'яті (в кожній строчці можуть бути дані із фіксованого набора адрес), що значно скорочує час пошуку. З кожною коміркою *RAM* може бити зв'язано більше однієї строчки кеш-пам'яті:

наприклад, *n*-канальна асоціативність (*n-Way set associative*) означає, що інформація по деякому адресу оперативної пам’яті може зберігатися в *n* місцях кеш-пам’яті (**але в кожен момент часу – тільки в одному місці із *n***).

Основоположним принципом логічного сегментування є той факт, що в межах любого окремо взятого сегмента тільки одна строчка може містити інформацію, що знаходиться по деякому адресу в оперативній пам’яті. Звідси витікає, що якщо адреса оперативної пам’яті відома, то контролеру кеша із всього сегмента слід обробити тільки одну строчку. Таким чином, при отриманні запита на читання або запис контролер опитає тільки потрібну строчку у всіх сегментах на предмет присутності (*cache hit*) або відсутності (*cache miss*) інформації, що відповідає отриманій з *TLB* буфера фізичній адресі. Або всі сегменти повернуть сигнал відсутності, або за виключенням одного, що відповість сигналом присутності (див. рис. 2.7 та 2.8).

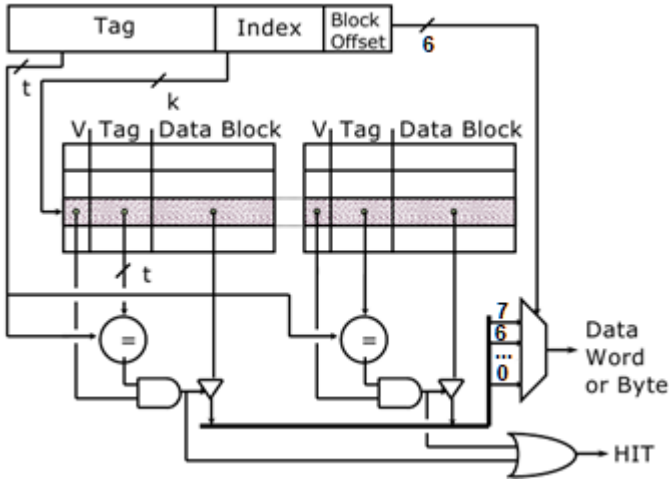

Рисунок 2.7 – Кеш-пам’ять з *2*- канальною асоціативністю

Якщо всі сегменти відповіли на запит сигналами відсутності інформації *Cache miss* (*HIT=0*), то запит до кеша *L1D* за локальною змінною (чи *L1I*) буде перенаправлено до контролера кеш-пам’яті більш низького рівня і так далі аж до контролера оперативної пам’яті. При цьому контролеру кеш-пам’яті потрібно вибрати сегмент кеш-пам’яті, куди буде здійснена підкачка потрібного блока даних з кеш-пам’яті більш низького рівня, або оперативної пам’яті. Для цього

він аналізує *valid bit* потрібної строчки у всіх сегментах кеш-пам’яті (число сегментів визначається асоціативністю кеш-пам’яті), після чого, відповідно до стратегії заміщення блоків даних, приймає рішення про збереження поточного значення вибраного блока даних з його наступним заміщенням на потрібний блок.

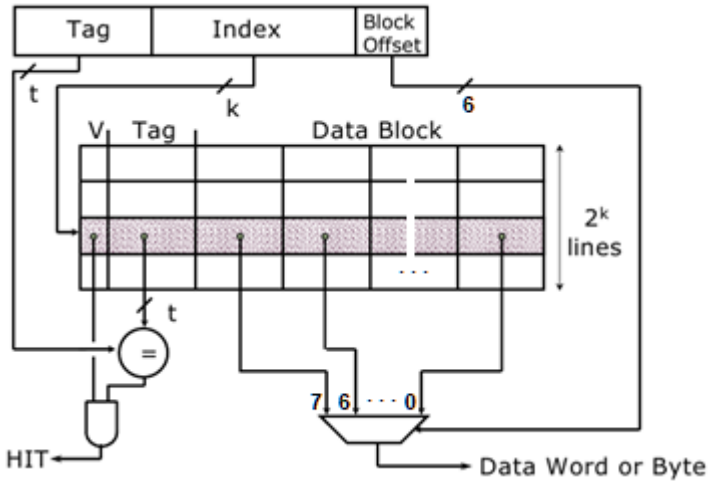

Рисунок 2.8 – Кеш-пам’ять з прямим відображенням

Як показали дослідження розробників фірми *Intel* [1]: **подвоєння асоціативності, при переході від кеш-пам’яті з прямим відображення до кеш-пам’яті з 2-канальною асоціативністю, або від 2-канальної асоціативності до 4-канальної асоціативності, має приблизно такий самий ефект на підвищення частоти звернень, як подвоєння розміру кешу. Однак збільшення асоціативності понад чотири не покращує показник попадання настільки сильно, і зазвичай це робиться з інших причин.** Деякі процесори можуть динамічно зменшувати асоціативність своїх кеш-пам’ятей у режимах низького енергоспоживання, що діє як захід енергозбереження.

Для вибору, який рядок асоціативної кеш-пам’яті із набору слід вилучити, коли набір стає повним, існують наступні **політики заміни**: випадковий; найменш використовуваний (*Least Recently Used – LRU*) (стан кешу *LRU* має оновлюватися при кожному доступі, реальна реалізація можлива лише для кешів з невеликою кількістю каналів асоціативності); двійкове дерево псевдо-*LRU*,

часто використовується для 4-8 каналів асоціативності; першим увійшов, першим вийшов (*First In, First Out – FIFO*), також відомий як *Round-Robin*, який використовується у високоасоціативних кешах; не останнє використаний *(Not Least Recently Used – NLRU)*; *FIFO* за винятком останнього використаного блоку.

Розрив між швидкістю процесора і швидкістю основної пам'яті зростає в геометричній прогресії. В останні роки швидкість процесора, виміряна його тактовою частотою, щорічно зростала на 55%, тоді як швидкість пам'яті зростала лише на 7% щорічно. Ця проблема відома як **стіна пам'яті** [1]. Мотивація створення такої структури, як кеш-пам’ять, та її ієрархії полягала в тому, щоб подолати цей розрив у швидкості та подолати стіну пам’яті.

У зв’язку з цим, вибір різних параметрів, таких як: розміру кешу, рівня його асоціативності, політики заміни є дуже критичними при розробці нової мікроархітектури і зазвичай старанно моделюються.

2.2.10 Механізми когерентності кешів (*Cache coherence*)

У комп’ютерній архітектурі **узгодженість кешу** (*cache coherence*) – це однаковість даних спільного ресурсу, які зрештою зберігаються в кількох локальних кешах. У *SMP* з окремою підсистемою кеш-пам’яті для кожного ядра може бути одночасно багато копій спільних даних (блоків даних): копія в основній пам’яті, копія в *L3* та копії в локальних кешах *L2* і *L1D* кожного ядра, яке їх запитало. Коли одна з копій даних змінюється, інші копії повинні відображати цю зміну. Узгодженість кешу гарантує, що зміни в значеннях спільних даних поширюються системою своєчасно.

Нижче наведені вимоги до когерентності кешів:

- **Розповсюдження запису.** Зміни даних у будь-якому кеші повинні поширюватися на інші копії (цього рядка кешу) у однорангових кешах.
- **Серіалізація транзакцій.** Теоретично когерентність може бути виконана на рівні завантаження/збереження. Однак на практиці це зазвичай виконується на рівні деталізації блоків кешу. Зчитування/запис в одну область пам’яті мають бачити всі ядра в одному порядку.

- **Визначення.** Когерентність визначає поведінку читання та запису в одну адресу.

- **Записи в одне й те саме місце мають бути послідовними.** Іншими словами, якщо фізична адреса *X* в *RAM* отримала два різних значення *A* і *B*, у цьому порядку, від будь-яких двох ядер, інші ядра ніколи не зможуть прочитати операнд з *X* як *B*, а потім прочитати його як *A*.

Найпоширенішим механізмом забезпечення узгодженості є **відстеження** (***snooping***). Вперше він був представлений в 1983 році [4]. *Snooping* – це процес, у якому окремі кеш-пам'яті відстежують адресні рядки для доступу до місць пам'яті, які вони кешують. Найпоширенішим протоколом, який використовує цей механізм, є протокол **недійсності запису** (***Write-invalidate***) [4].

Сутність цього протоколу полягає в тому, що коли ядро записує в спільний блок даних в кеші *L1D*, усі копії цього блоку даних в інших кешах стають недійсними через відстеження шини. Це гарантує, що тільки дійсна копія даних може бути прочитана та повторно записана ядром. Усі інші копії, в кешах інших ядер, стають недійсними та повинні бути перезаписані дійсним блоком даних, при виникненні в ядрі читання з цього блоку. До цієї категорії належать протоколи когерентності кешів: *MSI, MESI, MOSI, MOESI* та *MESIF*.

Реалізація протоколу *Write-invalidate* передбачає, що кожен рядок кешу має, як мінімум, два додаткові біти: *V* – дійсний; *D* – брудний біт, означає, що дані в кеші не збігаються з даними в пам'яті. Кожен рядок кешу перебуває в одному з таких станів: «брудний» (був оновлений локальним ядром), «дійсний», «недійсний» або «спільний». Рядок кешу містить блок даних, який можна прочитати або записати. Запис у рядок кешу змінює його стан. Коли блок вперше завантажується в кеш, він позначається як «дійсний». Стан «дійсний» означає, що рядок кешу є поточним. Під час локального запису до «дійсного» рядку кешу *L1D* (запис блока даних за фізичним адресом в *RAM*), стеження за шиною гарантує, що стан будь-які копії цього блоку даних в інших кешах встановлюються на «недійсний». «Недійсний» означає, що копія існувала в кеші, але тепер вона є не актуальною.

Коли система записує дані в кеш, вона повинна в якийсь момент також записати ці дані в резервне сховище (кеш нижчого рівня, або *RAM*). Час цього запису контролюється так званою політикою запису. Існує два основних підходи до запису в резервне сховище: **наскрізний запис** (***Write-through***) – запис виконується синхронно як до кешу, так і до резервного сховища; **зворотний запис** (***Write-back***) (також називається відкладеним записом) –спочатку запис виконується лише в кеш, а запис у резервне сховище затримується (за час затримки змінений вміст блоку даних може бути змінено іншим записом).

2.2.10.1 Протокол *MESI*

Протокол *MESI* є протоколом узгодженості кешу на основі *Write-invalidate* і *Write-back*. Він також відомий як Іллінойський протокол (через його розробку в Університеті Іллінойсу). Кеші зворотного запису можуть значно заощадити пропускну здатність, яка при наскрізному записі витрачається на запис в резервне сховище при здійснені запису в кеш. У кешах зворотного запису завжди присутній брудний стан, який вказує на те, що дані в кеші відрізняються від даних в резервному сховищі. Іллінойський протокол вимагає передачі блоку даних з кешу в кеш у разі промаху, якщо блок знаходиться в іншому кеші, задіяному в відстеженні. Цей протокол зменшує кількість транзакцій до резервного сховища порівняно з протоколом *MSI*. Це означає значне покращення продуктивності.

На рис. 2.9 показані біти *valid* та *dirty,* що мають бути присутніми в кожному рядку кешу для ідентифікації станів протоколу *MESI:*

- ***Exclusive (E), V=1, D=0.*** Блок даних присутній лише в кеші поточного ядра ("Ексклюзивний"), але він є чистим – він відповідає резервному сховищу. Коли інше ядро здійснює запит шини до основної пам’яті за цим же блоком даних – кеш поточного ядра поміщає значення копії блоку даних на шину, скасовуючи доступ до пам’яті. При цьому стани відповідних рядків в кешах обох ядер змінюються на "Спільний" *(Shared).* Якщо запис в цей рядок кеша поточного

ядра відбувається до розглянутої події, його стан переходить в "Змінений" *(Modified)*.

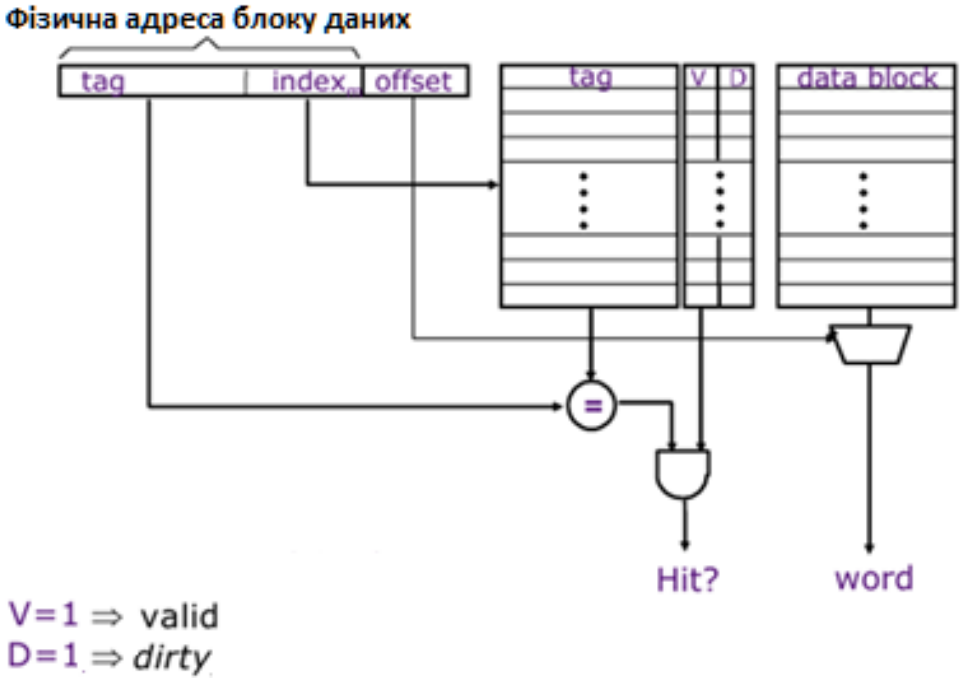

Рисунок 2.9 – Біти кодування станів в протоколі *MESI*.

- ***Shared (S), V=1, D=0.*** Вказує на те, що цей рядок кешу може зберігатися в кешах інших ядер мікропроцесора та є чистим – він відповідає резервному сховищу. Рядок може бути скасованим (зміненим на "Недійсний" *(Invalid)* стан) у будь-який час при виконанні операції запису у цей же блок даних в іншому ядрі мікропроцесора. Коли ще одне ядро здійснює запит шини до резервного сховища за цим же блоком даних – один із кешів, що вже його мають (за вибором розробника [10, 11]), поміщає значення копії блоку даних на шину, скасовуючи доступ до резервного сховища.
- ***Modified (M), V=1, D=1.*** Рядок кешу присутній лише в кеші поточного ядра та є брудним – його було змінено, він відрізняється від значення в резервному сховищі. Стан *S* рядку кешу, що містив цей же блок даних, у всіх інших ядрах було змінено на стан *I* ("Недійсний"). Кеш поточного ядра є відповідальним за зворотний запис блоку даних цього рядка назад до резервного сховища в якийсь час у майбутньому. До виникнення запиту шини до резервного сховища за цим

блоком даних у якомусь іншому ядрі, блок даних у поточному ядрі може неодноразово змінюватися, при цьому стан рядку кешу залишається незмінним *(M)*. Зворотний запис змінює стан рядку на "Спільний" (S). Він здійснюється паралельно з розміщенням значення копії блоку даних на шину у відповідь на запит іншого ядра на доступ до основної пам’яті за цим блоком, скасовуючи доступ до резервного сховища.

- ***Invalid (I), V=0, D=x.*** Вказує, що цей рядок кешу є "Недійсний" (не використовується).

Дозволені стани рядку кешу, що містять якийсь блок даних, для будь-якої пари кешів приведені на рис. 2.10 [11].

Як видно з цього рисунку, коли блок позначено як *M* або *E*, копії блоку в інших кешах позначені як I.

Якщо безперервні операції читання та запису певного блоку даних виконуються кешами різних ядер, потрібно щоразу здійснювати зворотній запис до резервного сховища. Але це не обов’язкова вимога, а лише додаткові накладні витрати, спричинені використанням *MESI*. Цю проблему вдалося подолати за допомогою протоколу *MOESI*.

	M	E	S	I
M	✗	✗	✗	✓
E	✗	✗	✗	✓
S	✗	✗	✓	✓
I	✓	✓	✓	✓

Рисунок 2.10 – Дозволені стани рядку кешу для будь-якої пари кешів в протоколі *MESI*.

Ще один недолік протоколу *MESI* полягає в тому, що коли якийсь блок даних одночасно перебуває у кешах декількох ядер і відповідні строчки кешів мають

стан "Спільний" (*S*), всі вони можуть розмістити значення копії блоку даних на шину у відповідь на запит ще одного ядра на доступ до резервного сховища за цим блоком. Стан *F* у протоколі *MESIF* вирішує цю надмірність (попередній недолік *MESIF* також вирішує).

2.2.10.2 Буфер зберігання *(Store Buffer)*

Буфер зберігання використовується під час запису в "Недійсний" рядок кешу [9]. Оскільки запис все одно триватиме, ядро видає повідомлення про недійсне читання (при цьому рядки кешу всіх інших ядер, які зберігають цю адресу пам’яті, набувають статусу "Недійсний"), а потім надсилає запис у буфер зберігання для виконання, коли рядок кешу нарешті надходить до кешу. Прямим наслідком існування буфера зберігання є те, що коли ядро здійснює запис, цей запис не відразу записується в кеш.

Таким чином, щоразу, коли ядру потрібно прочитати рядок кешу, воно спочатку сканує власний буфер зберігання на наявність того самого рядка, оскільки існує ймовірність, що той самий рядок був записаний тим же ядром раніше, але ще не був записаний у кеш (попередній запис все ще очікує в буфері зберігання). Зауважте, що в той час як ядро може читати власні попередні записи в буфері зберігання, інші ядра не можуть бачити ці записи, доки вони не будуть записані в кеш - ядро не може сканувати буфер зберігання інших ядер.

2.2.10.3 Протокол *MOESI*

Протокол *MOESI* розроблений *AMD* [12]. Це повний протокол когерентності кешів, який охоплює всі можливі стани, які зазвичай використовуються в інших протоколах. На додаток до чотирьох загальних станів протоколу *MESI* існує п’ятий стан "Власність" (*Owned*), який представляє як змінені, так і спільні дані. Це дозволяє уникнути необхідності записувати змінені дані в резервне сховище перед тим, як надати до них спільний доступ. Хоча зрештою дані все ще повинні бути записані, зворотний запис може бути відкладено. Для того, щоб це стало можливим, має бути можлива пряма передача даних з кеша одного ядра в кеш

іншого ядра, тому кеш із даними в модифікованому стані може надавати ці дані іншому зчитувача без перенесення їх у резервне сховище. Дозволені стани рядку кешу, що містять загальний блок даних, для будь-якої пари кешів приведені на рис. 2.11 [12].

	M	O	E	S	I
M	✗	✗	✗	✗	✓
O	✗	✗	✗	✓	✓
E	✗	✗	✗	✗	✓
S	✗	✓	✗	✓	✓
I	✓	✓	✓	✓	✓

Рисунок 2.11 – Дозволені стани рядку кешу для будь-якої пари кешів в протоколі *MOESI*

Modified ("Змінений"). Цей кеш містить єдину дійсну копію рядка кешу, і в цю копію внесені зміни.

Owned. ("Власність"). Цей кеш є одним із кількох із дійсною копією рядка кешу, але має виключне право вносити зміни до нього. Він повинен транслювати ці зміни в кеші усіх інших ядер, які спільно використовують блок даних. Введення стану "Власність" дозволяє обмінюватися "брудними" даними, тобто модифікований блок кешу можна переміщувати між різними кешами без оновлення резервного сховища. Рядок кешу із станом "Власність" може змінити свій стан на "Змінений" після того, як він зазнав змін, а усі спільні копії попереднього значення цього рядку у кешах інших ядер стали "Недійсними", або змінити свій стан на "Спільний" шляхом здійснення зворотного запису до резервного сховища. Рядки кешу у стані "Власність", мають відповідати даними на запит стеження.

Exclusive. Цей кеш містить єдину копію рядка, але рядок є "чистим" (незміненим).

Shared. Рядок є однією з кількох копій у системі. Цей кеш не має дозволу на зміну копії. Інші ядра в системі також можуть зберігати копії даних у стані

"Спільний". На відміну від протоколу *MESI*, рядок у такому стані може бути "забрудненим" щодо пам’яті; якщо це так, деякий кеш має копію у стані "Власність", і цей кеш відповідає за остаточне оновлення резервного сховища. Якщо жоден кеш не містить відповідний рядок у стані "Власність", то цей рядок є "чистим". Такий рядок кешу можна не записувати до резервного сховища. Його стан можна змінити на "Ексклюзивний" або "Змінений" стан після визнання "Недійсними" всіх копій у стані "Спільний". Рядки кешу у стані "Спільний" не відповідають на запити даних по шині стеження.

Invalid. Цей блок є "Недійсний"; його потрібно отримати, щоб задовольнити будь-яку спробу доступу.

Цей протокол, більш складна версія простішого протоколу *MESI*, дозволяє уникнути необхідності записувати "брудний" рядок кешу назад в резервне сховище, коли інше ядро намагається його прочитати. Натомість стан *Owned* дозволяє ядру передавати змінені дані безпосередньо іншому ядру. Хоча *MOESI* може швидко поділитися "брудними" рядками кешу, він не може швидко поділитися "чистими" рядками з кешу. Якщо рядок кешу "чистий" щодо пам’яті та перебуває у стані "Спільний", тоді будь-який запит стеження до цього рядка кешу заповнюватиметься з резервного сховища, а не з кешу.

2.2.10.4 Протокол *MESIF*

Протокол *MESIF* [13], розроблений *Intel* для когерентних архітектур кешу. Протокол складається з п'яти станів: *Modified (M), Exclusive (E), Shared (S), Invalid (I)* and *Forward (F)*.

Стани *M, E, S* та *I* такі ж, як і в протоколі *MESI*. Стан *F* є спеціалізованою формою стану *S* і вказує на те, що кеш має діяти як призначений відповідач на будь-які запити для даного рядка. Протокол гарантує, що якщо будь-який кеш зберігає рядок у стані *S*, щонайбільше один (інший) кеш утримує його в стані *F*.

У системі кеш-пам’яті, що використовує протокол *MESI*, запит на рядок кешу, отриманий декількома кешами, що містять рядок у стані *S*, обслуговуватиметься неефективно. Він може бути задоволений із (повільного) резервного сховища,

або всі спільні кеші можуть відповісти, бомбардуючи запитувача надлишковими відповідями. У системі кеш-пам’яті, що використовує протокол *MESIF*, на запит рядка кешу відповість лише кеш, який утримує рядок у стані *F* (див. рис. 2.12 [13]). Це дозволяє запитувачу отримувати копію зі швидкістю кеш-кеш.

	M	E	S	I	F
M	✘	✘	✘	✓	✘
E	✘	✘	✘	✓	✘
S	✘	✘	✓	✓	✓
I	✓	✓	✓	✓	✓
F	✘	✘	✓	✓	✘

Рисунок 2.12 – Дозволені стани рядку кешу для будь-якої пари кешів в протоколі *MESIF*

Оскільки кеш може в односторонньому порядку відхилити (зробити недійсним) рядок у станах *S* або *F*, можливо, що жоден кеш не матиме копії у стані *F*, навіть якщо копії у стані *S* існують. У цьому випадку запит на рядок задовольняється (менш ефективно, але все ж коректно) з резервного сховища. Щоб мінімізувати ймовірність того, що рядок *F* буде відкинутий через відсутність інтересу, останньому запитувачу рядка призначається стан *F*. Коли кеш у стані *F* відповідає, він передає стан *F* новому кешу. Таким чином, основна відмінність від протоколу *MESI* полягає в тому, що запит на копію рядка кешу для читання завжди надходить у кеш із цим рядком у стані *F*. **Стан *F* у протоколі *MESIF* – це просто спосіб вибрати одного з учасників чистого рядка кешу для відповіді на запит читання даних за допомогою прямої передачі з кешу в кеш замість очікування, поки дані надійдуть з резервного сховища. Ця оптимізація має сенс в архітектурах, де затримка між кеш-пам’яттю набагато менша порівняно з затримкою доступу до резервного сховища.** Ключовим моментом, на який слід звернути увагу, є те, що, подібно до протоколу

MESI, коли дані перебувають у загальному стані (з рядком в одному із кешів у стані *F*), дані є чистими.

2.3 Принципи розробки пристрою передбачення галуження

Конвеєрні процесори досягають максимальної пропускної спроможності, коли вони знаходяться в режимі стійкого безперервного потоку (*in the streaming mode*). Для стадії вибірки, потоковий режим означає продовження вибірки команд із послідовних комірок пам'яті *thread*.

Семантика потоку керування *thread* задається у формі графа потоку керування (ГПК), в котрому вершинами є основні операційні блоки (безумовні та умовні оператори), а дугами - переходи між головними операційними блоками у відповідності з заданим потоком керування. Наприклад, на рис. 2.13 a) [14] показаний ГПК з чотирьома основними операційними блоками, поміщеними в прямокутники, що виділені пунктиром.

При відображенні ГПК на лінійну форму послідовних байтів строк кеша *L1I* він змінюється за допомогою добавлення до нього команд безумовного переходу, як показано на рис. 2.13 b) [14] .

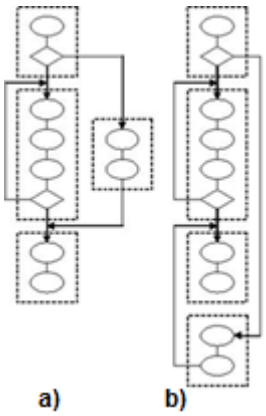

Рисунок 2.13 – Потік керування *thread*: a) граф потоку керування (ГПК); b) відображення ГПК на послідовні байти строк кеша *L1I*

При відображенні ГПК на лінійну форму послідовних байтів строк кеша *L1I* він змінюється за допомогою добавлення до нього команд безумовного переходу, як показано на рис. 2.13 b).

Кожна з команд розгалуження призводить до порушення безперервного режиму роботи конвеєра команд. Команди, що зв’язані з викликом метода, також не можуть вибиратися, дока не визначена адреса переходу. При умовних розгалуженнях, крім того, необхідно ще визначити напрямок переходу.

Обробка умовного розгалуження викликає затримку, котра призводить до холостих тактів в роботі конвеєра команд: декодуванням *CISC* на *RISC*, диспетчеризацією *RISC* та їх виконанням. Реальні втрати продуктивності, це число холостих тактів в роботі конвеєра команд, помножене на ширину паралельної видачі *CISC* команд на декодування (5-6 в *Sunny Cove* за такт, з урахуванням *Macro Fusion*).

Для режиму адресації відносно програмного лічильника, цільова адреса безумовного галуження може формуватися під час стадії вибірки. При цьому на наступному такті обробки цього *thread, CISCs* для декодування будуть вибиратися, починаючи з адреси безумовного галуження.

Якщо в потоці команд використовується режим непрямої регістрової адресації (посилання), команда переходу повинна була б пройти через стадію декодування и обробки, щоб отримати доступ до регістра, де знаходиться адреса переходу. Для регістрової непрямої адресацій із зміщенням, до адреси переходу потрібно ще додати зміщення.

Для умовних розгалужень потрібно було б ще врахувати затримку при формуванні архітектурного стону, перевірка котрого складає напрямок розгалуження.

Для досягнення максимальної пропускної здатності конвеєра команд суперскалярного ядра необхідно мінімізувати число холостих непродуктивних тактів в його роботі, або максимально використати ці такти для організації якоїсь корисної роботи. Для цього, для всіх вищезгаданих випадків розгалуження, котрі могли б привести до затримок в декодуванні *CISCs* потоку, був розроблений метод передбачення галужень, котрий часто носить назву "спекулятивна обробка галужень", або "спекулятивні обчислення".

2.3.1 Використання цільового буферу розгалуження

Кеш під назвою *Branch Target Buffer (BTB)* зберігає цільову адресу всіх *jump*$_s$ (див див. рис. 2.14 [14]). Цільова адреса зберігається в *BTB* під час першого безумовного переходу або першого умовного переходу. Під час другого виконання того самого переходу цільова адреса в *BTB* використовується для отримання прогнозованої адреси вибірки наступної послідовності *CISC* в конвеєрі команд.

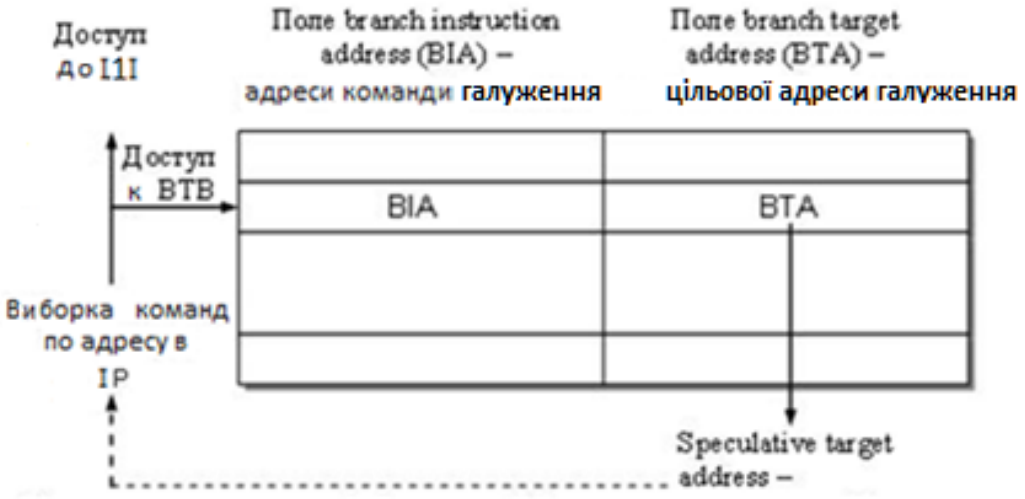

Рисунок 2.14 – Виконання галужень з використанням буфера *BTB*

Справжня адреса переходу обчислюється при досягнені етапу його виконання. Передбачена адреса, швидше за все, буде правильною для безумовних переходів, але не є певною, оскільки *BTB* може бути недостатньо великим, щоб вмістити всі такі переходи в програмі, тому різні переходи можуть замінювати записи один одного в *BTB*. Ризик помилкового прогнозування значно вищий для умовних переходів.

Експериментальні дослідження показали, що поведінка команд галуження дуже вірогідно передбачувана. В наслідку того, що команда галуження зазвичай неодноразово повторюється в програмі, її динамічна поведінка може бути відстеженою. На підставі минулої поведінки (виконання чи не виконання галуження) фактично може бути передбаченою її майбутня поведінка. В цьому контексті можливо виділити дві складові частини передбачення галужень -

branch target speculation **- спекулятивне передбачення цільового результату галуження** (передбачення цільової адреси галуження) та ***branch condition speculation*** **- спекулятивне передбачення результату умовного галуження**.

Для любого спекулятивного метода передбачення галужень повинні застосовуватися практичні засоби визначення достовірності передбачення та надійного повернення до первинного стану при помилковому спекулятивному передбаченню (*misprediction*).

2.3.2 Передбачення цільової адреси галуження (*branch predictor*)

Передбачення цільової адреси галуження основується на використанні *BTB* для запису цільових адрес недавно виконаних попередніх галужень. Цей буфер є малою спрощеною кеш-пам'яттю, доступною на протязі стадії вибірки наступної послідовності команд. Кожне надходження в *BTB* містить два поля (див. рис. 2.14 [14]): *branch instruction address (BIA)*- адреса самої команди галуження та *branch target address (BTA)* - цільової адреси галуження.

Коли команда галуження виконується вперше, в *BTB* для неї розміщується запис. Адрес команди галуження поступають в поле *BIA*, а цільова адреса галуження записується в поле *BTA*. *BTB* є повністю асоціативним кешем, поле *BIA* використовується для асоціативного доступу до *BTB*. До буфера *BTB* можливий доступ одночасно з вибіркою з *L1I*. Коли поточне значення покажчика команд (*IP* - див. підрозділ 1.6), враховуючи ширину вибірки паралельних команд, співпадає з адресом в *BIA*, відбувається *«влучання»* в *BTB*. Це означає, що одна із команд, котрі зараз вибираються із *L1I*, вже була раніше виконаною і є командою галуження. При «влучанні», із поля *BTA* вибирається цільова адреса наступної послідовності команд цього потоку команд і записується в *IP*.

При цьому, спекулятивна цільова адреса галуження буде готовою для використання її в наступному циклі вибірки *CISCs* цього *thread*. Якщо таке передбачення галуження виявляється вірним, то команда галуження фактично виконується на стадії вибірки, не викликавши ніяких вимушених циклів непродуктивних втрат в роботі конвеєра команд. На стадії завершення

виконання передбаченого галуження (в *Back-end* ядра мікропроцесора), коли умова переходу вже обчислена виконується перевірка чи було це передбачення вірним. Інакше трапилось помилкове передбачення (*misprediction*), котре потребує відновлення всіх архітектурних станів процесорного ядра, що мали місце до точки передбачення. Результати перевірки передбаченого галуження в *Back-end* використовуються для оновлення вмісту буфера *BTB (*поля *BTA).*

Існує декілька способів передбачення результату умови галуження. Простий спосіб полягає в розробці апаратної підтримки попередньої вибірки, при котрій завчасно передбачається, що галуження не виконається. Коли вибирається команда галуження, то ще до її виконання продовжується вибірка наступної послідовності команд. Коли становиться відомо, що умовне галуження виконується, то в програмний лічильник заноситься цільова адреса галуження. Зрозуміло, що така проста форма передбачення є неефективною. Наприклад, часто при організації програмних циклів галуження виконується, тоді на стадії вибірки чергової послідовності команд, що йдуть за командою умовного переходу, в *IP* потрібно підставляти цільову адресу галуження, за винятком виходу із циклу.

Іншим способом передбачення результату умовного галуження є використання додаткових програмних можливостей. Цей спосіб потребує включення в *ISA* (архітектуру системи команд) спеціальних керуючих ознак. Це означає, що в команди передачі управління необхідно включати спеціальні біти або поля. Наприклад, в формат команди галуження вводять додатковий біт, котрий установлюється компілятором. Цей біт використовується в якості підказки (ключа) для апаратури, котра передбачає галуження. Для цього компілятор, обробляючи команди галужень, створює профільну інформацію про хід виконання програми (*profiling information*). На основі цієї інформації він визначає найбільш вірогідне початкове значення для цього біта. Це забезпечує наявність в кожній команді галуження *«підказки»* про напрямок галуження. Таке передбачення є програмним, статичним, але сам хід виконання програми і

галужень має динамічний характер. Таке статичне програмне передбачення було використано, наприклад, в *Motorola 88110*.

Найбільш поширений спосіб передбачення результату умовного галуження, що широко використовується в сучасних суперскалярних процесорах, ґрунтується на статистичній передісторії виконання галужень. Спосіб полягає в запам'ятовуванні станів: *taken (T)* - галуження приймається, чи *not taken (N)* - галуження не приймається. Такий підхід вперше був запропонований *Jim Smith*, котрий його запатентував від імені *Control Data*. Спосіб ґрунтується на припущенні того, що накопичена інформація про напрямок галуження, може бути корисною для майбутніх виконань галуження. Проекті рішення для такого типу передбачення галужень містять кількісний показник попередньої історії, котра повинна відслідковуватися для кожного історичного інформаційного набору (*history pattern*), або регістра глобальної історії переходів (*GBHR*). На основі попередньої історії фактичного направлення галуження рішення про напрямок галуження приймає *finite state machine (FSM)* – кінцевий автомат станів, показаний на рис. 2.15 [14].

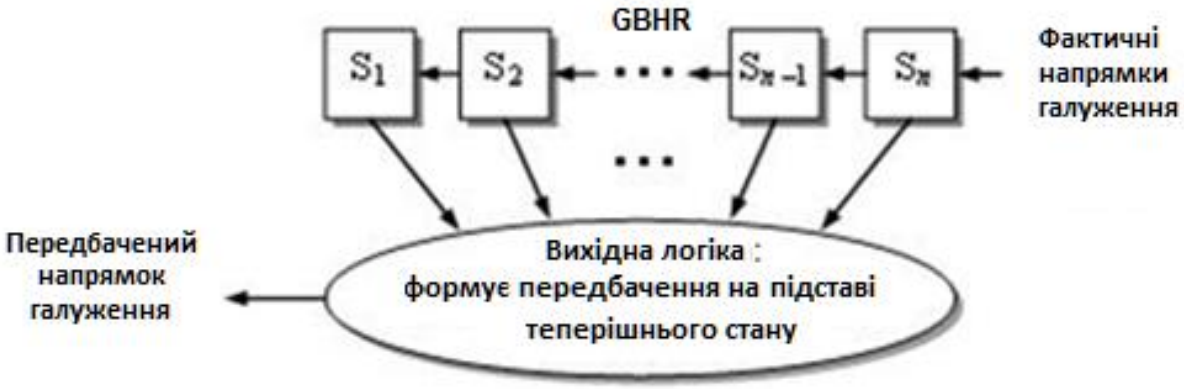

Рисунок 2.15 – Модель *FSM* для передбачення напрямку галуження на основі його попередньої історії

Стан *GBHR* кодує останні *n* напрямки цього як послідовність *taken* та *not taken (T,…, N)*. Вихідна логіка формує передбачення на основі поточного стану *FSM*. Коли передбачене таким чином галуження здійснюється, його фактичний результат, після підтвердження в *ROB*, використовується в якості вхідної змінної

в *FSM* для зберігання фактичного напрямку галуження. Таким чином, *GBHR* для кожного галуження – це зсувний регістр, в який записуються *n* останніх фактичних напрямків виконання цього галуження.

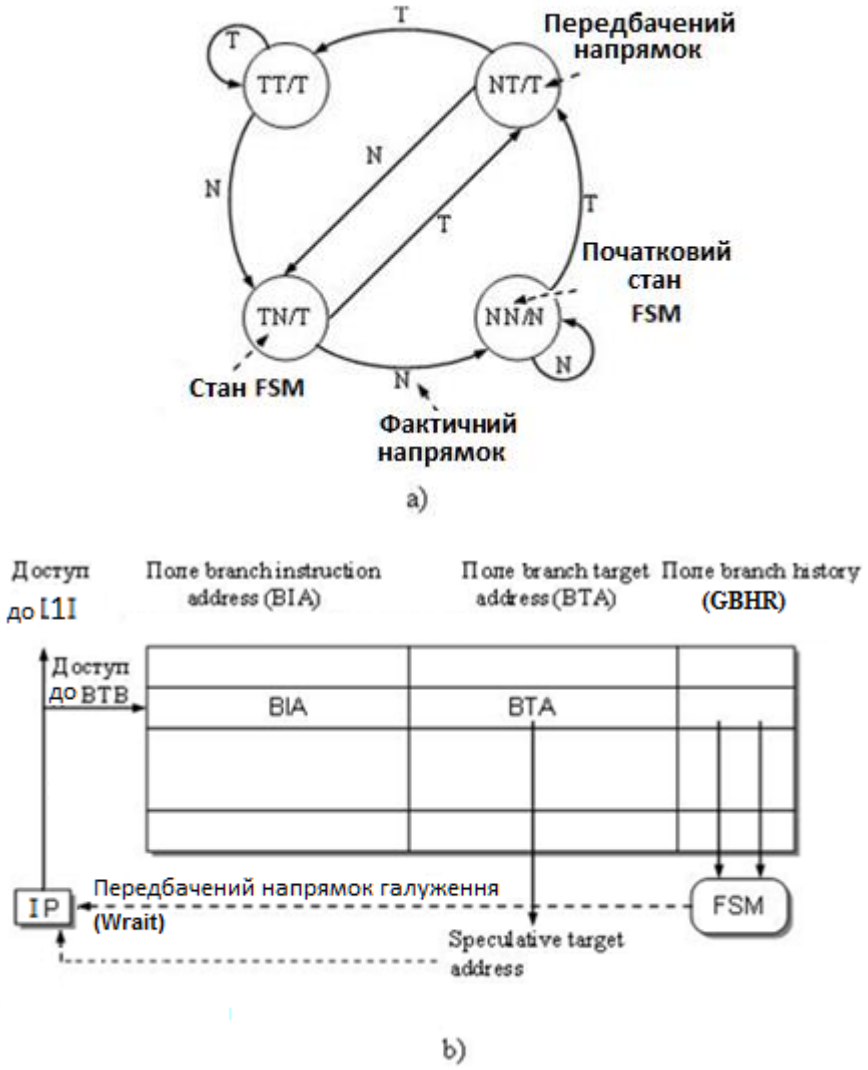

Рисунок 2.16 – Спосіб передбачення галужень, що ґрунтується на попередній історії виконаних галужень: a) - *FSM* з 2-розрядним пристроєм передбачення галуження; b) - буфер *BTB* з додатковим полем для запису бітів попередньої історії фактично здійснених галужень

На рис. 2.16 a) [14] показана схема переходів станів *FSM* 2-разрядного пристрою передбачення галуження, котрий використовує всього два біта попередньої історії, щоб здійснювати на їх основі передбачення галуження. Два біта попередньої історії визначають стан *FSM*.

Пристрій *FSM* може бути в одному із чотирьох можливих станів: *NN*, *NT*, *TT*, або *TN*, представляючи направлення двох попередніх фактично – виконаних

галужень. Стан *NN* може визначатися як початковий стан. Після підтвердження в *ROB* фактичного напрямку галуження відповідний біт *taken* чи *not taken* використовується в якості вхідної змінної в *FSM* для збереження напрямку цього галуження. При цьому відбувається зміна стану *FSM*, котрий відтворює попередню історію цього галуження. Ця попередня історія буде використовуватися для наступного передбачення цього галуження

На рис. 2.16 b) [14] показана схема буфер *BTB* з додатковим полем для запису бітів попередньої історії галужень.

В *Sandy Bridge,* у порівнянні з *Nehalem*, передбачення стали точнішими завдяки збільшенню довжини буфера *BTB* і подовженню регістра глобальної історії переходів (*GBHR*) [15].

В *Nehalem BTB* був присутнім у вигляді дворівневої ієрархії: малого - «шидкого» *L1* та значно більшого - «повільного» *L2* (по тій же причині, чому існують декілька рівнів кеша). Але в *Sandy Bridge* використовувався один рівень *BTB*, причому його розмір був вдвоє більшим, ніж *L2 BTB* у *Nehalem*, в той час як розмір кеша *L1I* співпадає у всіх *CPU Intel,* починаючи з *Pentium M* [15].

BTB у *Sandy Bridge* зберігав адреси в стисненому стані, що дозволяло мати вдвоє більше комірок при схожих: площі та споживанні [15].

Алгоритм передбачення у *Sandy Bridge* був не новий, а оптимізований старий – загальний для: непрямих переходів, циклів і умовних переходів [15].

Nehalem мав 18-бітний *GBHR* [15]. Подробиці реалізації *BTB* в *Sunny Cove* поки що невідомі.

2.4 Призначення *Back-end* суперскалярного ядра

По черзі для кожного із 2 *hardware threads*, при включеній технології *Hyper-Threading, RISC* – операції (*μops*) поступають на входи *Back-end* мікропроцесорного суперскалярного ядра, див. рис. 2.17, де показаний *Back-end Sunny Cove*. Робота *Back-end* ґрунтується на використанні удосконаленого алгоритму Томасуло (*Tomasulo's algorithm*) [14].

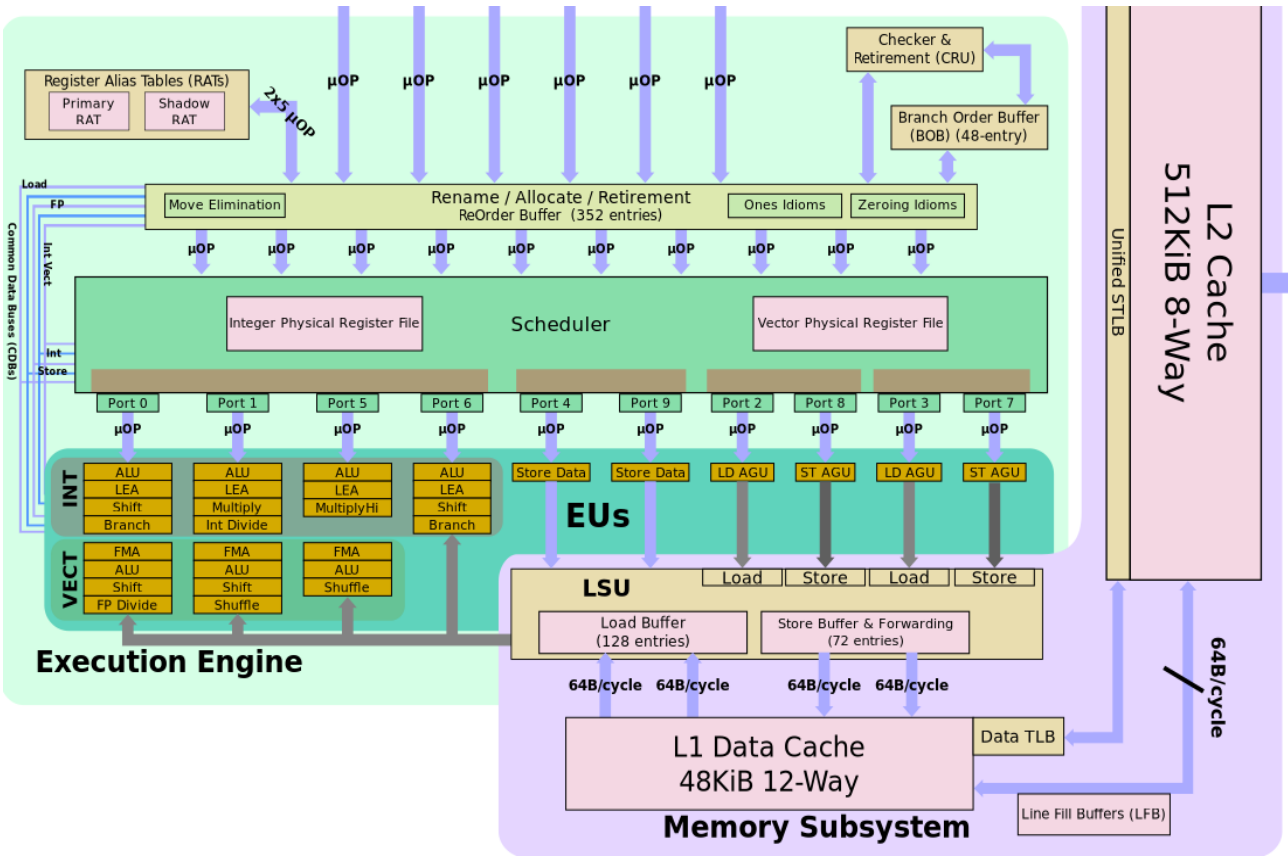

Рисунок 2.17 – *Back-end* мікропроцесорного суперскалярного ядра *Sunny Cove*

2.4.1 Покращений алгоритм Томасуло

Вперше алгоритм Томасуло був використаний в *IBM 360/91* [14] його розробником Робертом Томасуло в 1966 р. В той час, ще нічого не було відомо про архітектуру *dataflow*, котру *J. Dennis* винайшов в середині 1970-х років. Тому алгоритм Томасуло і не припускав використання *RS*, але він гуртувався на ідеї почергової видачі результатів з виходів виконуючих блоків процесора на загальну шину. Багато різних приймачів, що очікують один із результатів виконуючих блоків, можуть загрузити його з цієї шини **самостійно й одночасно**. В наступному він був удосконалений для використання *RDF* (див. підрозділ 1.3.2) в ядрі суперскалярного мікропроцесора.

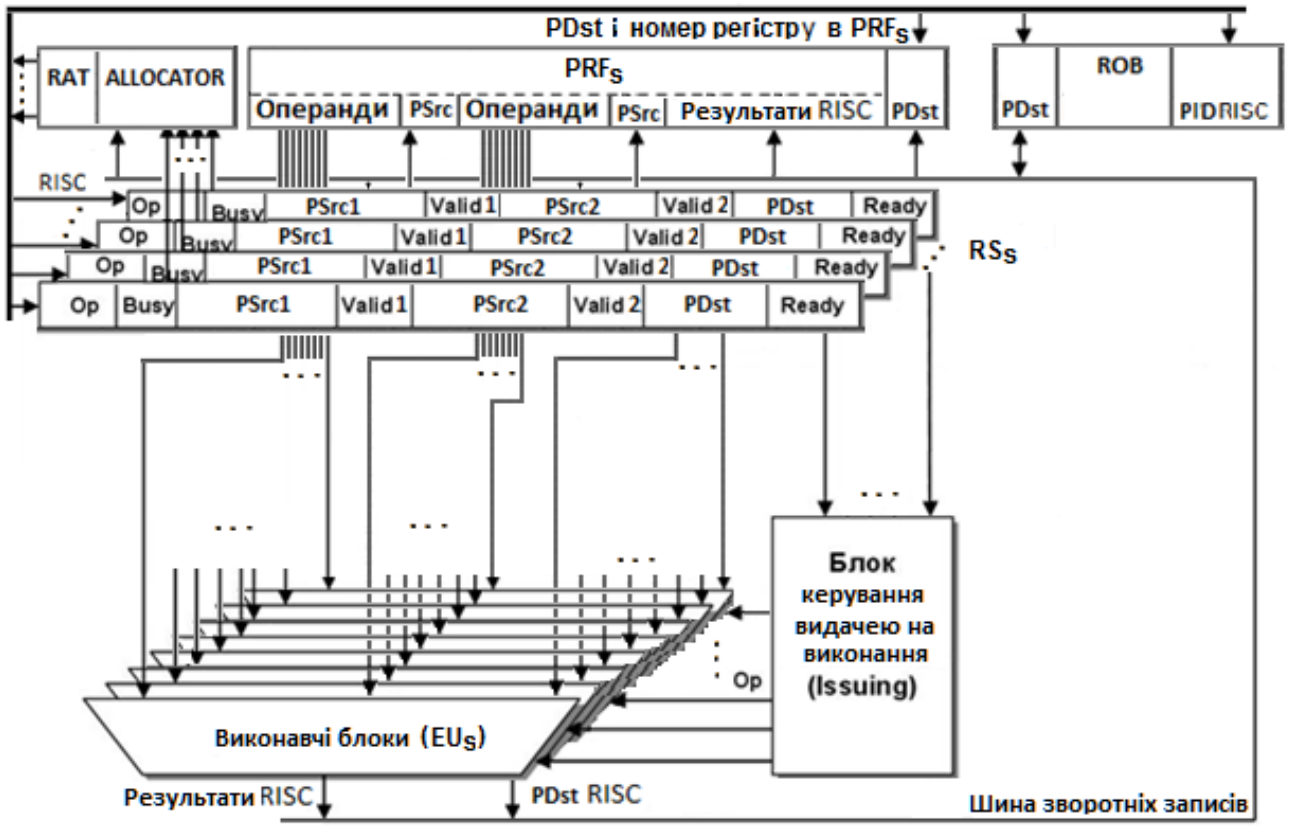

Рисунок 2.18 – Надходження *RISC*-операцій в *RS*, набуття *RISC*-операціями стану готовності до виконання та видача їх на обробку

Удосконалений алгоритм Томасуло ґрунтується на використанні станцій резервування (*Reservation stations – RS*, або по іншому активних комірок пам’яті (*Scheduler Entries*) для тимчасового зберігання декодованих *RISC* – подібних операцій (*μops*). *RISC* – операція зберігається в *RS*, із станом біта *Ready=0*, до моменту надходження її операндів (операнду) в перейменовані архітектурні регістри файлу регістрів загального призначення (див. рис. 2.18 [14]) і підтвердження завершення, відповідних *RISC* – операцій в *ROB* (біт *Valid1=1* та *Valid2=1*). При цьому *RS* становиться готовою до виконання і у відповідності з принципами *RDF*, формує сигнал для арбітра виконуючого блока (*Issuing*) про свою готовність до виконання (*Ready=1*).

За допомогою буферизації операндів (посилань *PSrc* на регістр в *Physical Register File-PRF*) в *RS* можливо запобігти зупинок конвеєра, зв’язаних з погрозами збоїв за даними: *WAR*, *WAW* та *RAW*.

2.4.2 Погрози збоїв за даними

У випадку, коли в сусідніх *RISC* – операціях *thread* для задавання джерела операнду або приймача результату використовується один і той же архітектурний регістр виникають погрозами збоїв за даними, котрі визиваються наступними трьома причинами:

- ***RAW- (read – after – write)*** погроза [14]: значення з відповідного регістра прочиталось раніше, ніж воно туди записалося (обновилося). Якщо запис значення операнду в регістр здійснено після раніше виконаного з нього читання, то це раніш прочитане значення є помилковим і програма в результаті буде виконана некоректно;
- ***WAR- (write - after - read)*** погроза [14]: нове значення операнду записалось в деякий регістр раніше, ніж з цього місця виконалось необхідне читання. Якщо команди виконуються в позачерговому порядку (***out - of - order***), то необхідно забезпечити читання попереднього (старого) значення операнду, до його заміни новим значенням операнду. В звичайному скалярному конвеєрному процесорі такої погрози немає, оскільки там команди виконуються по одній в програмному порядку. В суперскалярному процесорі команди виконуються паралельно і в позачерговому порядку, тому є погроза перезапису ще не зчитаного значення операнду;
- ***WAW- (write - after - write)*** погроза [14]: значення операнду правильно записалось в деякий регістр, але потім нове значення операнду знову записалося в теж місце (і це вже неправильно, якщо спочатку повинно виконатися читання попереднього значення).

Поняття погрози ***RAR (read - after - read)*** не існує, оскільки операції читання можуть проходити в довільному порядку.

В суперскалярному процесорі погрози збоїв за даними усуваються на архітектурному рівні шляхом використання **динамічного планування *POE (Plan Of Execution),* перейменуванням регістрів і завершення виконаних *RISC* – операцій в програмному порядку в *ROB (Reorder Buffer* - буфері переупорядкування)** [14]**.** Наприклад, в двох наступних *RISC* – операціях:

a=a+b; d=a-c погроза збоїв за даними усувається тим, що одному і тому ж архітектурному регістру *a* в першій *RISC* – операції будуть призначені різні фізичні регістри, тому для другої *RISC* – операції операнд *a* не буде готовим до тих пір, поки він не перевизначиться в іншому регістрі та його правильність не підтвердить *ROB.*

2.4.3 Станції резервування

Станції резервування (*RS*), або *Scheduler Entries* (див. табл. 1.2) являються місцем, де *RISC*-операції, що надходять від *Allocator* (пристрій виділення ресурсів ядра мікропроцесора)) [14], очікують надходження коректних операндів (поки не розрішаються залежності між *RISC*-операціями і стануть готовими операнди для їх виконання) та поки не стануть доступними (звільняться) відповідні виконуючі пристрої. В кожному циклі роботи конвеєрного ядра блоком керування вдачею на виконання (*Issuing* – див. рис. 2.18) здійснюється вибірка чергової групи готових до виконанню *RISC*-операцій. Одночасно, *RSs* можуть містити *RISC*-операції від різних потоків.

RISC-операції поступають в *RSs* від *RAT* (*Register Alias Table* - таблиці альтернативних імен регістрів (синонімів)) [14]. Номер *RS* для надходження *RISC*-операції виділяє *Allocator* на підставі аналізу бітів *Busy* в *RSs (Busy = 0*). При надходженні *RISC*-операції в *RS*, значення її біта занятості *Busy* змінюється на протилежне [14].

Кожна *RS* містить поля (див. рис. 2.18) для запису тегів джерел операндів *PSrc* (*PIDRISC Source* – фізичний номер регістра в *PRF*) і поле тегу *PDst* (*PIDRISC Destination* - фізичний номер регістра в *PRF* для запису результату). Код *PDst* разом з кодом *Op* і операндами передається на входи вибраного, за допомогою *Issuing,* виконуючого пристрою. Після виконання *RISC,* код *PDst* видається разом з результатом на шину зворотних записів і використовується для запису результату в регістр *PRF* (виділений *Allocator* для результату виконання *RISC* з кодом *PDst*). В *ROB* для *RISC* з кодом *PDst* відмічається факт її виконання, перевіряється чи відноситься вона до гілки з вірно передбаченим переходом та

здійснюється її завершення в програмному порядку – у відповідності до коду *PIDRISC,* посилаючи відповідне повідомлення в *Allocator* (код *PIDRISC* містить інформацію до якого потоку відноситься ця *RISC* і про порядок декодування *CISC* на *RISC* в цьому потоці).

2.4.4 Перейменування регістрів

Логіка перейменування регістрів перейменовує архітектурні *x86-64* регістри в машинні фізичні регістри [14]. Це дозволяє 16 скалярним та 32 (16) векторним регістрам загального призначення в архітектурі *x86-64* кожного з оброблюваних *hardware threads* динамічно розширюватися, відображаючись на масиви доступних, відповідно, 180 та 168 фізичних регістрів у *Skylake* (див. табл. 1.2). Ця логіка використовує ***Register Alias Table (RAT)*** для установлення актуальної версії набору архітектурних регістрів, щоб закодувати в *RISC*-операції посилання *PSrc* на фізичні регістри операндів та посилання *PDst* на фізичний регістр, де має бути розміщено результат виконання *RISC*.

Оскільки кожен логічний процесор повинен підтримувати і відстежувати свій повний стан архітектури, в кожний момент часу є дві актуальні таблиці тимчасових імен регістрів *RATs*, по одній для кожного логічного процесора, якщо ядро підтримує технологію *Hyper-Threading* (див. підрозділ 1.4). Процес перейменування регістрів в них проводиться паралельно та відповідно до роботи логіки виділення ресурсів, що носить назву *Allocator*. Таким чином, логіка перейменування регістрів в *RISC*-операціях використовує фізичні регістри, для яких *Allocator* установлює ресурси (кожному значенню *PDst* ставить у відповідність номер фізичного регістру).

Використання *RAT* забезпечує перейменування використовуваних в процесі обчислень архітектурних регістрів в фізичні регістри. Це дає можливість усунення залежностей за даними між *RISC*-операціями і забезпечує більш високий рівень *ILP*.

Принцип перейменувань регістрів для одного *kernel thread* за допомогою *RAT* показано на рис. 2.19 [14].

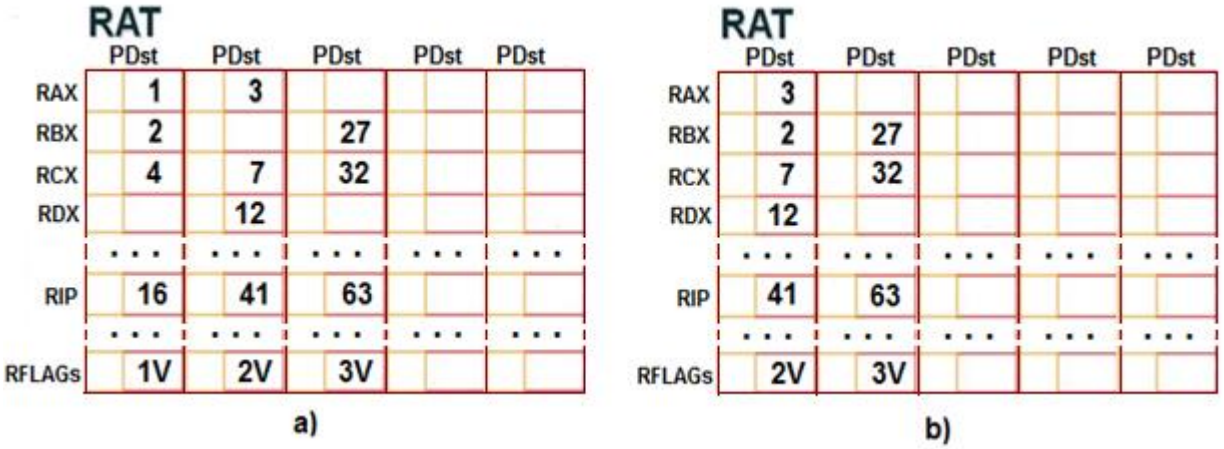

Рисунок 2.19 – Принцип перейменування регістрів за допомогою *RAT* для одного з оброблюваних потоків *kernel thread*: a)-архітектурні стани регістрів потоку на момент двох «спекулятивних» передбачень; b)-архітектурні стани регістрів потоку на момент, коли *ROB* виявив, що перше передбачення було не «спекулятивним» й завершив усі *RISC* у відповідному *hardware thread*

Коли *ROB* виявив, що відбулося неправильне прогнозування галуження, з *RAT Allocator* повинен видалити непотрібні регістри, які були обчислені у цьому *hardware thread,* передати їх до переліку вільних регістрів, та знову повернутися до відображень архітектурних регістрів на фізичні в наступному наборі *RISC*-операцій.

При неправильно прогнозованому галуженні всі незавершені *RISC*-операції, що ще очікують свого виконання в *RSs* і мають посилання на неправильно прогнозовані фізичні регістри, стають недійсними та звільняють *RSs (Busy = 0)*.

2.4.5 Пристрій призначення ресурсів *(Allocator)*

У кожному тактовому циклі *Allocator* здійснює виділення ресурсів, необхідних для обробки *RISC* що надходять, та надсилає відповідні повідомлення в: *ROB, RAT, RS, PRFs,* буфер завантажень (*LB – Load Buffer*), а також в буфер запам’ятовувань (*SB – Store Buffer*) [14]. Для цього він декодує *RISC*-операції, що надходять від *Allocation Queue*, для визначення потрібних для них ресурсів. У тому випадку, коли *Allocator* не може здійснити виділення потрібних ресурсів для чергової порції *RISC*-операцій, подальша видача *RISC*

призупиняється, поки не стають доступними достатньо ресурсів для подальшого виконання шляхом вилучення з обробки попередніх *RISC*.

Для *ROB*: *Allocator* відводить одне вільне ***Out-of-Order Window*** [14] (див. табл. 1.2) для кожної з *RISC*, що надходить. При цьому в спеціальні поля *Out-of-Order Window* записується *PDst RISC* та *PIDRISC* (ідентифікатор *RISC*-операції, що відображає до якого *hardware thread* вона відноситься та порядок декодування *CISC* на *RISC* в цьому *hardware thread*). Якщо *ROB* заповнений, *Allocator* повинен встановити сигнал призупинення роботи *Allocation Queue* для запобігання перезапису стану незавершеної *RISC* у *ROB*.

Для *RAT*: *Allocator* призначає один вільний фізичний регістр з *PRFs* для кожної з *RISC*, що надходить та заносить його до *RAT* відповідного *kernel thread*. Ці призначення використовуються *RAT* для створення відповідних записів в таблиці альтернативних імен архітектурних регістрів.

Для RSs: *Allocator* призначає номер вільної *RS* для кожної з *RISC*, що надходить для обробки. Якщо вільної *RS* не має, то *Allocator* виставляє призупинення роботи *Allocation Queue* для запобігання перезапису дійсних даних в *RSs*. Для кожної *RISC*, що надходить, *Allocator* використовує *RAT* відповідного *kernel thread* для кодування перейменованих архітектурних регістрів операндів (*PSrc1* та *PSrc2*) та перейменованого архітектурного регістру результату (*PDst*).

Оскільки *RSs* диспетчеризують *RISC*-операції для їх виконання в позачерговому порядку – по мірі готовності їх даних, записи *RISC*-операцій, що надходять до *RSs*, розподіляються відповідно до порядку звільнення цих комірок від раніше записаних в них *RISC*-операцій. Для цього використовується схема бітової матриці (побітового відображення), де кожен запис, що поступає в *RS* відображається на біт в накопичувачі розміщень в *RS* (біт *Busy* - див. рис. 2.18). Таким чином, записи в *RS* можуть заміщуватися у будь-якому порядку. *Allocator* шукає вільні *RSs*, скануючи бітову матрицю від місця завершення виділення для попередньої порції *RISC*-операцій і поки не знаходить потрібну кількість перших вільних *RSs*.

2.4.6 Буфер переупорядковування (*ReOrder Buffer*)

ReOrder Buffer (ROB) складається з комірок, що носять назву *Out-of-Order Window* [14]. З кожною стадією «Так» в розвитку *CISC-RISC* архітектури мікропроцесорів фірми *Intel* їх кількість постійно збільшується: 168 у *Sandy Bridge*, 192 у *Haswell*, 224 у *Skylake*, *352* у *Sunny Cove* (див. табл. 1.2).

Список вільних комірок в *ROB* підтримує *Allocator* [14]. Він визначає адресу вільної *Out-of-Order Window* в *ROB* для кожної з *RISC*, що плануються до виконання та записує в неї *PDst RISC* та *PIDRISC*. Призначення відбуваються послідовно від 0 і до найвищої адреси, а потім пошук вільної адреси для призначення знову починається з 0.

ROB застосовується в мікроархітектурі *RISC*-ядра суперскалярного процесора для завершення *RISC* у програмному порядку (відповідно до номера *PIDRISC*, що *RISC* набувають його при декодуванні) [14]. При визначенні апаратурою ядра циклу (див. *LSD* в підрозділі 2.1.4), декодування *CISC* на *RISC* призупиняється. При цьому *PIDRISC* з циклу *Allocator* змінює на новий *PIDRISC*, продовжуючи нумерацію у порядку декодування [14].

ROB перевіряє кожну *RISC*, що зв'язана з галуженням. Визначає *BTA* галуження та фактичний напрямок галуження і передає їх, відповідно, до *BTB* та *GBHR* (див. підрозділ 2.3.2).

Ніяка *RISC*, після її виконання, не є завершеною поки її не завершить *ROB* [14], привівши у відповідність всі архітектурні стани шляхом надсилання в *Allocator* повідомлення про її завершення та чи була вона вірно прогнозованою, а також *ROB* сповіщає всі *RSs*, що очікують результат виконання *RISC* з конкретним *PDst* у якості свого *PSrc1* чи *PSrc2* про готовність операнду. Завдяки завершенню *RISCs* у порядку збільшення їх *PIDRISC*, не дивлячись на їх паралельне та позачергове виконання, досягається ефект їх послідовного виконання відповідно до лічильника команд [14].

ROB містить сліди усіх виконуваних *RISC*-операції від усіх виконуваних в даний момент *hardware thread*, у тому числі й службового потоку диспетчера (а в нульовому ядрі ще й службового потоку планувальника), що знаходяться в

виконавчому конвеєрі. Тобто *CISCs*, які були декодовані, але ще архітектурно не завершилися. Вони включають усі *RISC*-операції, що знаходяться: в *RSs*, на стадії виконання в масиві *EU*, а також ті *RISC*-операції, що вже завершили своє виконання та їх результати знаходяться у *PRFs*, але *ROB* (див. рис. 2.20) ще не завершив їх виконання [14].

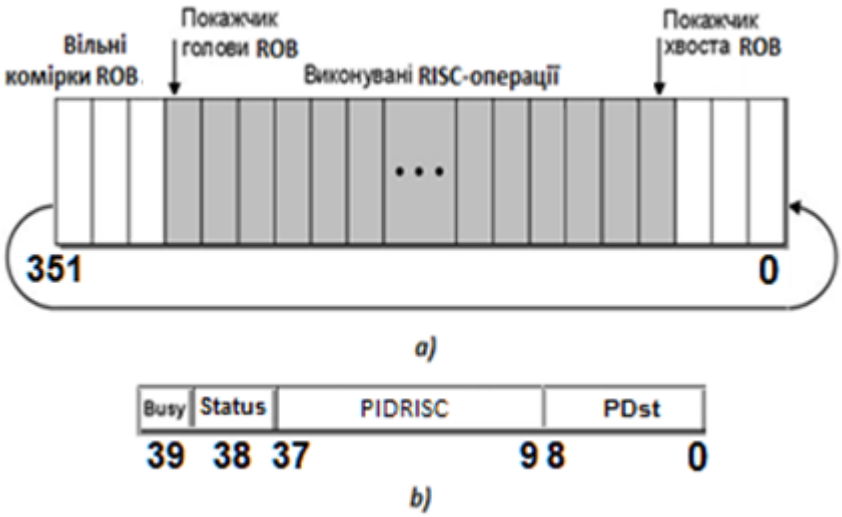

Рисунок 2.20 – Структура *ROB*: a)- завершення і вилучення *RISC* з *ROB*, починаючи з його «голови», та надходження нових *RISC* в *ROB*, починаючи з його «хвоста»; b)-структура комірки *ROB*.

Стан кожної *RISC* у *ROB* відстежується за допомогою біту *Status* [14]. При розпізнаванні на шині зворотних записів (див. рис. 2.18) коду *PDst* біт *Status=1*, що свідчить про виконання *RISC* в масиві *EU* та запис результату її виконання в фізичний регістр з номером *PDst*. При цьому *ROB* може переходити до її завершення, після завершення всіх *RISC* з меншим значенням *PIDRISC* [14].

3 ІЄРАРХІЯ ПАМ’ЯТІ КОМП’ЮТЕРА

На теперішній час авторові відомі часові характеристики для ієрархії пам’яті комп’ютера побудованого на ядрі *Coffee Lake* (2018), 14 нм:

- читання із файлу фізичних регістрів *(PRF)* - 16 за цикл*;*
- записи до *PRF-8* за цикл*;*
- *L1I* - 4 циклів*;*
- *L1D* - 4-5 циклів (*5* для складних адрес)*;*
- *L2* - 12 циклів*;*
- *L3* - 42 цикли*;*
- *RAM* - 60 циклів*;*
- *SSD* через *NVMe/PCIe* - $n10^2$ циклів (де *n=1, 2, 3*);
- *SSD* через *SATA3* - 10^3 циклів;
- *HDD* - $m10^4$ циклів (де *m=5, 6,..., 12*).

В регістри загального призначення *PRF* дані загружаються з *L1D*. У випадку їх відсутності там, блок даних (64Б), що відноситься до *heap,* може бути зчитаним в *L1D* з *L1D* іншого ядра (див. протокол *MESIF* в підрозділі 2.2.10.4). Якщо цього блока даних в *L1D* інших ядер немає, або потрібен блок даних, що відноситься до *stack* (див. підрозділ 1.6), то запит буде направлено в наступний рівень кеш-пам’яті і так далі, поки дані не будуть доставлені в *L3* з *RAM (Random-Access Memory).*

3.1 Модулі пам’яті для побудови *RAM*

Використання для побудови *RAM* комірок статичного типу *(SRAM - Static Random-Access Memory)* є економічно невиправданим (відомі технології побудови комірок з 6 польових транзисторів, або 4 транзисторів та 2 резисторів), тому майже завжди перевага надається коміркам динамічного типу *(DRAM - Dynamic Random-Access Memory)*, кожна з котрих складається всього з 1 польового транзистора і 1 конденсатора. Логічне значення такої комірки визначається по напрузі заряду конденсатора. Час зарядки/розрядки

конденсатора комірки *DRAM* вище, ніж час відкриття/закриття каналу польового транзистора комірки *SRAM*, і вже тільки цей фактор визначає більш низьку швидкодію *DRAM* у порівнянні з *SRAM*. Крім того, із-за неминучих витоків току в конденсаторах їх необхідно періодично перезаряджати, тобто зчитувати вміст комірок і записувати прочитане назад. Люба операція читання з комірки *DRAM* приводить до втрати конденсатором свого заряду. *DRAM* дешевша ніж *SRAM*, але в той же час і значно повільніша. Для подолання проблем з затримкою сигналу в *DRAM* стали вбудовувати невелику кількість *SRAM*, тобто створювати на чипі кеш. Така *RAM* стала називатися *SDRAM*. Подальший розвиток технології побудови *SDRAM* привів до появи пам'яті *DDR SDRAM* (*Double Data Rate SDRAM* – *SDRAM* з подвоєною швидкістю передачі даних). В *DDR SDRAM* досягається подвоєння швидкості роботи, у порівнянні з *SDRAM*, в наслідку зчитування інформації не тільки по фронту, як в *SDRAM*, але і по спаду тактового сигналу. При цьому, подвоєння швидкості передачі даних досягається без збільшення частоти тактового сигналу шини пам'яті [ ] (див. рис. 3.1).

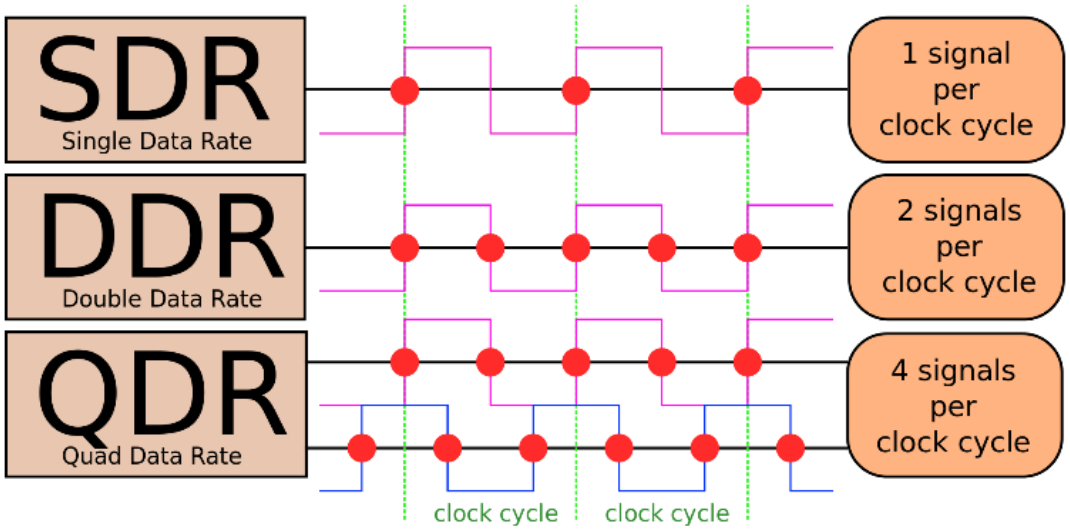

Рисунок 3.1 – Подвоєння швидкості передачі даних досягається без збільшення частоти тактового сигналу шини пам'яті, збільшення вчетверо – з подвоєнням частоти тактового сигналу

Для простоти таку пам'ять часто називають просто *DDR*. В наступному були розроблені модулі пам'яті: *DDR2, DDR3, DDR4, DDR5* котрі і використовуються в теперішньому часі для побудови *RAM*. В той же час, пам'ять: *DDR6* и *DDR7* в

теперішньому часі вже використовується для побудови відеоадаптерів. Для модулів пам’яті *DDRi* (із зростанням *i*), характерним є поступове: збільшення частоти тактового сигналу шини пам’яті та зниження напруги живлення.

В найгіршому випадку, якщо підсистема керування віртуальною пам’яттю операційної системи встигла витіснити їх з оперативної пам’яті в *swap*-файл (на жорсткому диску) втрати часу на доставку необхідних даних для регістра загального призначення ядра складуть десятки тисяч тактів роботи ядра.

3.2 Багатоканальність оперативної пам’яті

У сферах цифрової електроніки та комп’ютерного обладнання багатоканальна архітектура пам’яті – це технологія, яка збільшує швидкість передачі даних між пам’яттю *DRAM* і контролером пам’яті шляхом додавання більше каналів зв’язку між ними. Теоретично це множить швидкість передачі даних на кількість наявних каналів. Двоканальна пам'ять використовує два канали. Цей метод бере свій початок ще в 1960-х роках і використовувався в *IBM System/360 Model 91* і в *CDC 6600*. Сучасні процесори високого класу, як-от серії *Intel i7 Extreme* та *AMD Ryzen Threadripper*, а також різноманітні *Xeon* підтримують чотириканальну пам’ять.

3.2.1 Двоканальна архітектура

Двоканальні контролери пам'яті в системній архітектурі ПК використовують два 64-розрядні канали даних. Двоканальний режим не слід плутати з подвійною швидкістю передачі даних (*DDR*), у якій обмін даними відбувається двічі за такт *DRAM*. Ці дві технології не залежать одна від одної, і багато материнських плат використовують обидві, використовуючи пам’ять *DDR* у двоканальній конфігурації.

Двоканальна архітектура (див. рис. 3.2 [16]) потребує двоканальної материнської плата та двох або більше модулів пам’яті.

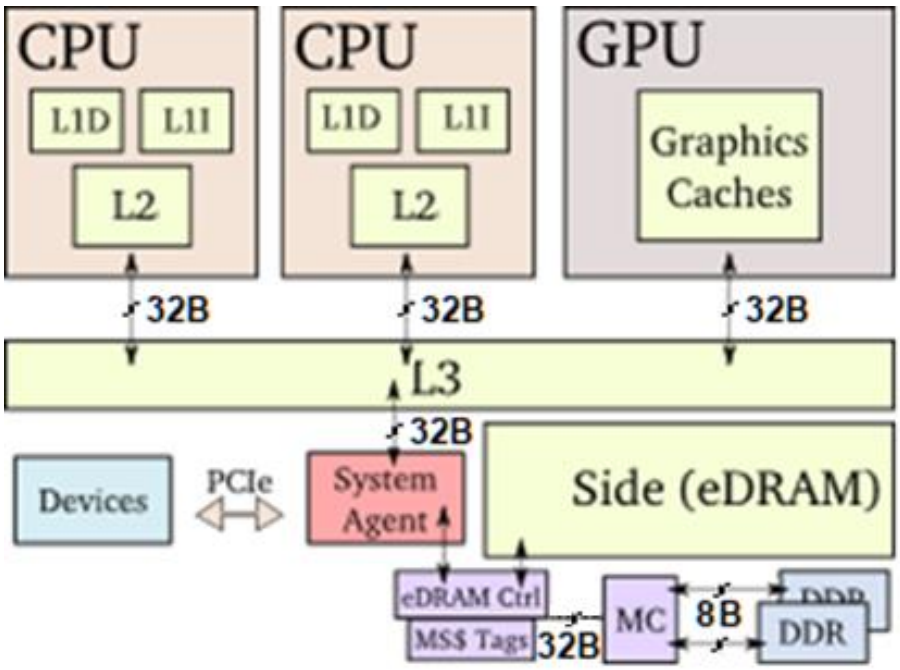

Рисунок 3.2 – Двоканальна архітектура пам’яті в *Skylake*

Кожен доступ до пам’яті *Skylake*, який проходить через контролер пам’яті, шукається у вбудованій *DRAM (eDRAM)* – динамічній пам’яті з довільним доступом, яка зазвичай інтегрована в логічну інтегральну схему або як частина *SoC*, або як окремий чіп, упакований разом із основною *SoC* [16]. При задоволеному попаданні значення отримується звідти. У разі промаху, значення доставляється з модулів основної пам’яті та зберігається в *eDRAM*.

Модулі пам'яті встановлюються у відповідні банки, які зазвичай позначаються кольором на материнській платі. Ці окремі канали надають контролеру пам’яті доступ до кожного модуля пам’яті. Не потрібно, щоб модулі пам’яті були ідентичними, але це часто рекомендуються для найкращої двоканальної роботи.

Материнські плати, що підтримують двоканальне розташування пам’яті, зазвичай мають кольорові роз’єми *DIMM*. Схеми кольорів не стандартизовані та мають протилежне значення у різних виробників, залежно від їх намірів та фактичного дизайну материнської плати. Збіги кольорів можуть вказувати на те, що гнізда належать одному каналу (це означає, що пари *DIMM* мають бути встановлені в гнізда різного кольору), або вони можуть використовуватися для

вказівки на те, що пари *DIMM* мають бути встановлені в той самий колір (це означає, що гнізда однакового кольору належать до різних каналів). У інструкції користувача материнської плати буде надано пояснення, як встановити пам'ять для цього конкретного пристрою. При наявності різних модулів пам'яті, для кращої роботи основної пам'яті, пару модулів пам'яті однакової ємності потрібно розмістити в першому банку кожного каналу, а пара модулів різної ємності – у другому банку.

Модулі з різною швидкістю можна запускати в двоканальному режимі, хоча материнська плата тоді запускатиме всі модулі пам'яті на швидкості найповільнішого модуля. Проте деякі материнські плати мають проблеми з сумісністю з певними марками чи моделями пам'яті під час спроби використання їх у двоканальному режимі. З цієї причини, як правило, рекомендується використовувати ідентичні пари модулів пам'яті, тому більшість виробників пам'яті продають «набори» модулів *DIMM* із відповідними парами. Кілька виробників материнських плат підтримують лише конфігурації, у яких використовується «відповідна пара» модулів. Відповідна пара має відповідати:

- Ємність. Деякі набори мікросхем *Intel* підтримують мікросхеми різної ємності (такий режим роботи називається *Flex Mode*): ємність, яку можна порівняти, працює в двоканальному режимі, тоді як решта працює в одноканальному.
- Швидкість. Якщо швидкість не однакова, використовуватиметься менша швидкість двох модулів. Так само використовуватиметься більша затримка двох модулів.
- Так само затримка *CAS (CL)*, або строб адреси стовпця.
- Кількість чипів і задіяних сторін на модулях пам'яті (наприклад, дві сторони по чотири чіпи на кожній стороні).
- Відповідний розмір рядків і стовпців в чипах модулів пам'яті.

Двоканальна архітектура – це технологія, яка реалізується на материнських платах виробником і не поширюється на модулі пам'яті. Теоретично, будь-яка

відповідна пара модулів пам’яті може використовуватися в одноканальній або двоканальній роботі за умови, що материнська плата підтримує таку архітектуру.

Теоретично, двоканальна конфігурація вдвічі збільшує пропускну здатність пам’яті порівняно з одноканальною. Це не слід плутати з пам’яттю *DDR*, яка подвоює використання шини *DRAM*, передаючи дані по обом фронтах тактових сигналів шини пам’яті.

Експериментальні дослідження показали, що двоканальна архітектура забезпечувала, у найкращому разі, 5% збільшення швидкості завдань, що потребують пам’яті. Тест із застосуванням *SiSoftware Sandra* показав приблизно 70% збільшення продуктивності чотириканальної конфігурації порівняно з двоканальною конфігурацією. Теоретичне подвоєння пропускної здатності основної пам’яті, що працює в двоканальному режимі, не сильно впливає на реальне підвищення швидкості виконання додатків у зв’язку з тим, що в добре спроектованому ядрі процесора більшість звернень до пам’яті реалізуються, як звернення до кешів першого рівня: *L1I* та *L1D*.

Двоканальний режим спочатку був задуманий як спосіб максимізації пропускної здатності пам’яті шляхом об’єднання двох 64-бітних шин в одну 128-бітну шину (режим *«g*anged*»*). Це ретроспективно називається «спареним» режимом. Однак через незначний приріст продуктивності в додатках користувачів більш сучасні реалізації двоканального режиму за замовчуванням використовують режим *«unanged»* («без змін»), який підтримує дві 64-розрядні шини пам’яті, але дозволяє незалежний доступ до кожного каналу, підтримуючи багатоканальністю пам’яті багатопотоковий режимом роботи процесорного ядра та багатоядерність.

Приклади процесорів, що підтримують двоканальну архітектуру пам’яті на модулях *DDR5*: *Alder Lake (2021); Raptor Lake (2022)*.

3.2.2 Триканальна архітектура

Триканальна архітектура пам’яті на модулях *DDR3* використовувалася у серії *Intel Core i7-900*. Платформа *LGA* 1366 (наприклад, *Intel X58*) підтримувала

триканальну архітектура пам’яті на модулях *DDR3*, що зазвичай працювали на частотах 1333 і 1600 МГц, але могла працювати й на вищих тактових частотах на деяких материнських платах. У процесорах *AMD Socket AM3* не використовувалася триканальна архітектура *DDR3*, а лише двоканальна архітектура. Те саме стосується серій *Intel Core i3*, *Core i5* та *Core i7-800*, які використовувалися на платформах *LGA 1156* (наприклад, *Intel P55*).

За інформацією від *Intel, Core i7* з *DDR3*, що працює на частоті 1066 МГц, забезпечить пікову швидкість передачі даних 25,6 ГБ/с при роботі в потрійному режимі з чергуванням каналів. Це призводить до швидшої продуктивності системи, а також більшої продуктивності на ват. Під час роботи в триканальному режимі затримка пам’яті зменшується завдяки чергуванню, тобто доступ до кожного модуля здійснюється послідовно для менших бітів даних, а не повністю заповнюється один модуль перед доступом до наступного. Дані розподіляються між модулями по черзі, потенційно втричі збільшуючи доступну пропускну здатність пам’яті для того самого обсягу даних, на відміну від зберігання всіх в одному модулі. Цю архітектуру можна використовувати лише тоді, коли всі три (або кратні трьом) модулі пам’яті однакові за ємністю та швидкістю і розміщені в триканальних слотах. При встановленні двох модулів пам'яті архітектура працюватиме в режимі двоканальної архітектури.

Приклади *CPUs*, що підтримують цю архітектуру пам’яті: *Intel Core i7 9xx: Bloomfield, Gulftown; Intel Core i7-9x0X Gulftown; Intel Xeon: E55xx Nehalem-EP, E56xx Westmere-EP, ECxxxx Jasper Forest, L55xx Nehalem-EP, L5609 Westmere-EP, L5630 Westmere-EP, L5640 Westmere-EP, LC55x8 Jasper Forest, Wxxxx Westmere-EP.*

3.2.3 Чотириканальна архітектура

Чотириканальна архітектура пам’яті на модулях *DDR4* замінила триканальну архітектуру на модулях *DDR3.* Вона реалізована на платформі *Intel X99 LGA 2011*, а також використовується в платформі *AMD Threadripper*. Чотириканальна архітектура *DDR3* використовується в платформі *AMD G34* і в платформі *Intel*

X79 LGA 2011. Процесори *AMD* для платформи *C32* і процесори *Intel* для платформи *LGA 1155* (наприклад, *Intel Z68*) замість цього використовують двоканальну пам’ять *DDR3*.

Цю архітектуру можна використовувати лише тоді, коли всі чотири модулі пам’яті (або кратні чотирьом) однакові за ємністю та швидкістю та розміщені в чотириканальних слотах. При встановленні двох модулів пам'яті архітектура буде працювати в двоканальному режимі; при встановленні трьох модулів пам'яті архітектура буде працювати в триканальному режимі.

Приклади *CPUs*, що підтримують цю архітектуру: *AMD Threadripper 2nd gen: 2990WX, 2970WX, 2950X, 2920X; AMD Threadripper: 1950X, 1920X, 1900x; AMD Opteron: 6100-series "Magny-Cours", 6200-series "Interlagos", 6300 -series "Abu Dhabi»; Intel Core: i7-9800X, i9-7900X, i7-7820X, i7-6950X, i7-6900K.*

3.2.4 Шестиканальна архітектура

Підтримується серверними процесорами *Qualcomm Centriq* і процесорами *Intel Xeon Scalable.*

Серія продуктів *Centriq 2400* стала доступна виробникам серверів у 2017 році з шестиканальною пам’яттю *DDR4*. Це продукти *SoC*, розроблені компанією *Qualcomm* для центрів обробки даних. *CPU Centriq* використовує набір інструкцій *ARM RISC* із кількома ядрами *CPU* в одному чіпі.

Процесори *Intel Xeon Scalable* відноситься до ряду сімейств *Xeon* на основі останніх серверних процесорів *Intel*: *Xeon Bronze, Xeon Silver, Xeon Gold, Xeon Platinum.* Вони також стали доступна виробникам серверів у 2017 році з шестиканальною архітектурою пам’яті на модулях *DDR4*.

3.2.5 Восьмиканальна архітектура

Підтримується серверними процесорами *AMD Epyc* та *Cavium ThunderX2*.

Процесори *Epyc* з’явилися у 2017 році та призначені для тих же цілей, що і процесори *Intel Xeon Scalable.* Вони успішно конкурують з процесорами: *Xeon Bronze, Xeon Silver, Xeon Gold.* Усі поточні процесори *Epyc* оснащені

восьмиканальною архітектурою пам’яті на модулях *DDR4* на різних швидкостях. Ці процесори наступного покоління, як підтверджено *AMD*, будуть мати дванадцятиканальну архітектуру пам’яті на модулях *DDR5*.

Процесори *ThunderX2* – це сімейство 64-розрядних багатоядерних серверних мікропроцесорів *ARM*, представлених компанією *Cavium* на початку 2018 року, які замінили оригінальну лінію *ThunderX*. Вони орієнтовані на маршрутизатори, комутатори, сховища даних та сервери. Оснащаються восьмиканальною архітектурою пам’яті на модулях *DDR4*.

СПИСОК ЛІТЕРАТУРИ

1. John L. Hennessy, David A. Patterson. Computer Architecture: A Cuantitative Approach 6th Edition, USA, Morgan Kaufmann, 2019– 1527 p.
2. Intel® 64 and IA-32 Architectures Software Developer Manuals – URL: https://software.intel.com/en-us/articles/intel-sdm.
3. IEEE 754: Standard for Binary Floating-Point Arithmetic [Електронний ресурс] / 3 квітня 2014. – URL: http://grouper.ieee.org/groups/754/.
4. J. Shen, M. Lipasti. Modern Processor Design: Fundamentals of Superscalar Processors. Waveland Press, 2013 - 642 p.
5. Bonnell (microarchitecture) - Wikiwand.
6. Y. Patt, W. Hwu, et al, Experiments with HPS, a Restricted Data Flow Micro architecture for High Performance Computers, Digest of Papers, COMPCON 86, (March 1986), pp. 254-258.
7. http://www.intel.com/products/ht/hyperthreading_more.htm for information.
8. Stack frame layout on x86-64 - Eli Bendersky's website (thegreenplace.net).
9. Sunny Cove - Microarchitectures - Intel - WikiChip.
10. Cache coherence in shared-memory architecture. University of Machester. Cache Coherence (utexas.edu).
11. MESI protocol - Wikipedia.
12. MOESI protocol - Wikipedia.
13. MESIF protocol - Wikipedia.
14. Луцький Г. М. та ін. Закл. звіт по НДР № ДР 0213U004830, -Київ, 2013 - 165 с.
15. Sandy Bridge (client) - Microarchitectures - Intel - WikiChip.
16. Skylake - WikiChip.