Lecture 12:
CPU – Superscalar

Hunjun Lee

hunjunlee@hanyang.ac.kr

Our next stuff:
Can we make CPU faster?

- The latency of the CPU:
  - wall clock time = (time/cyc) x (cyc/inst) x ~~(inst/program)~~ 컴파일러가 수행 → 무시
    - time/cyc: the critical path delay
    - cyc/inst: the number of cycles to execute a instruction
    - inst/program: ISA and compiler determine this

How to make a faster CPU?
(1) Reduce time / cyc
(2) Increase inst / cyc → IPC ↑

ILP: Instruction-level parallelism

동시에 실행될 수 있는 Instruction의 사용

- ILP is the parallel or simultaneous execution of a sequence of instructions
  - Inter-dependent instructions cannot be executed in parallel

- Program ILP = Avg. # of instructions / Cycle (step)
  - How many instructions are simultaneously executed in parallel

Data dependency 때문에 따로 실행해야함
→ ILP=1

```
code1:
addi  $r1,  $r2,  1
divi  $r3,  $r1,  17
sub   $r4,  $r0,  $r3
```

Max ILP = 1
(execute serially)

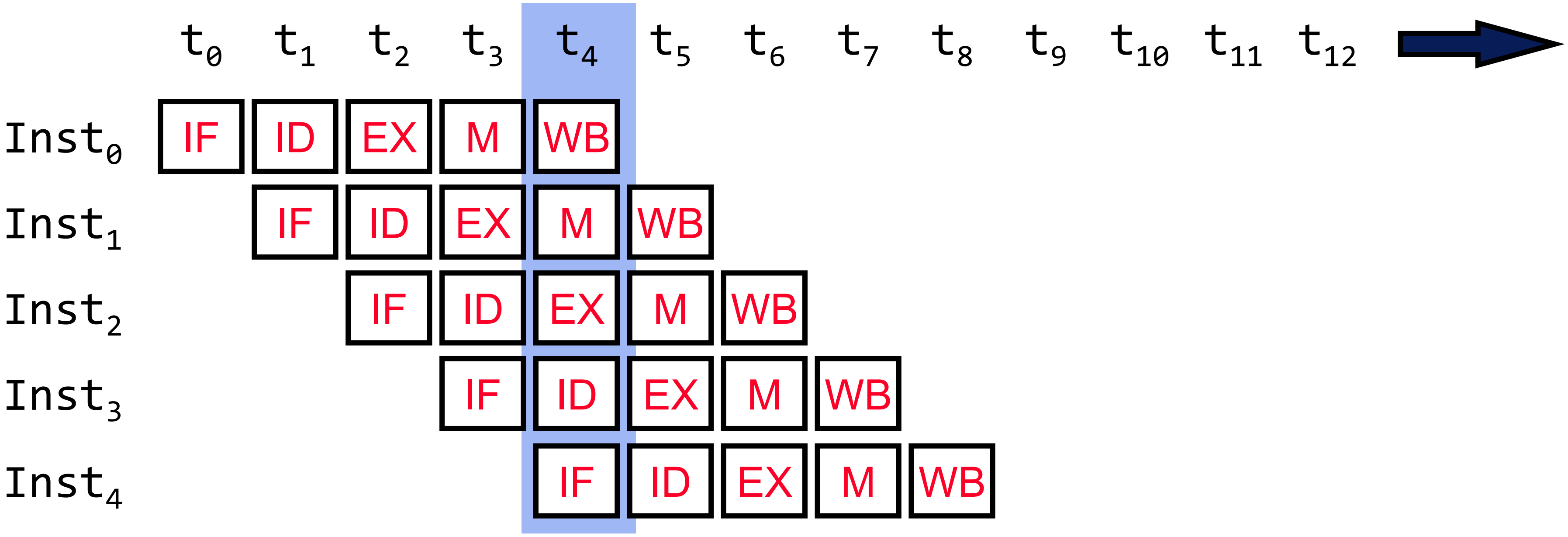
Data dependency X
→ 모든 순서에 실행가능
→ ILP ≤ 3

```
code2:
addi  $r1,  $r2,  1
divi  $r3,  $r9,  17
sub   $r4,  $r0,  $r10
```

Max ILP = 3
(execute parallel)

How to exploit ILP: Pipeline

- Pipelining: executing multiple instructions in parallel
  - **Operation latency =** 1
  - **Peak IPC =** 1
  - **HW ILP =** # of instructions / # of cycles required = 1 ILP = 1이다.
    - We parallelize instructions at the cost of #cycles

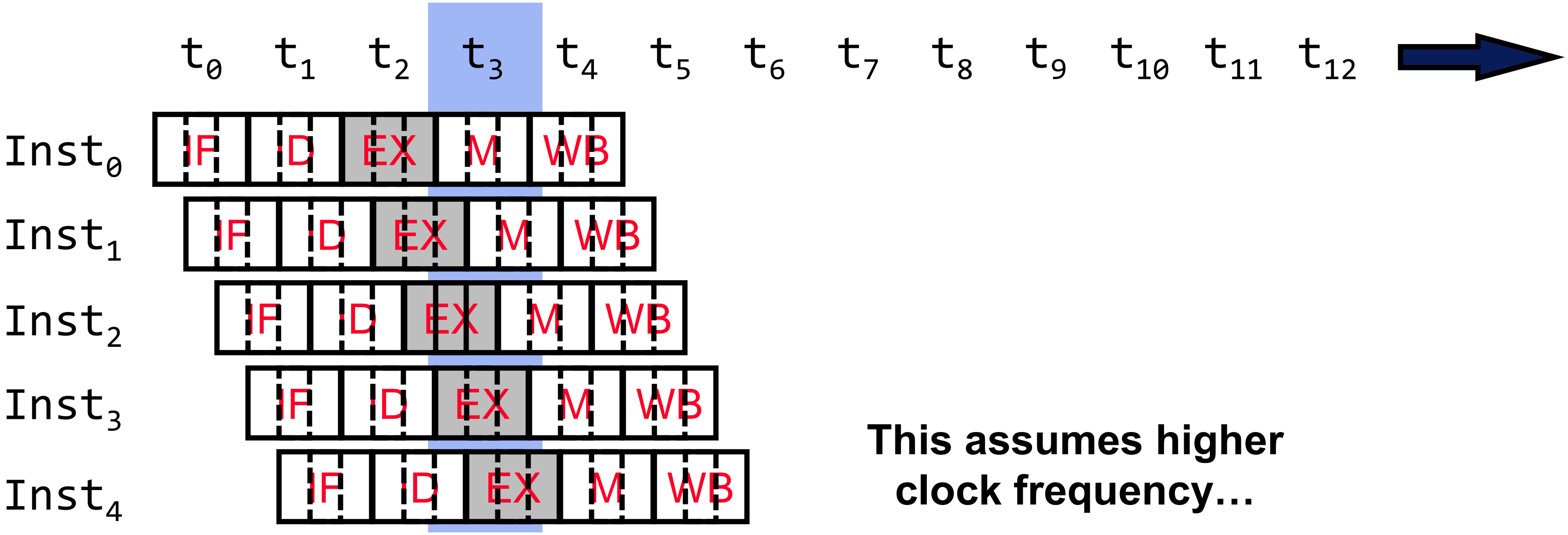


	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
Inst_0	IF	ID	EX	M	WB								
Inst_1		IF	ID	EX	M	WB							
Inst_2			IF	ID	EX	M	WB						
Inst_3				IF	ID	EX	M	WB					
Inst_4					IF	ID	EX	M	WB				

Make the pipeline deeper (superpipeline)

↳ 동시에 fetch되는 건 아니다.

- Superpipeline execution (split a cycle into M minor cycles):
  - **Operation latency =** 1 baseline cycle (M minor cycles)
  - **Peak IPC =** M per baseline cycle
  - **HW ILP =** # of instructions / # of (baseline) cycles required = M



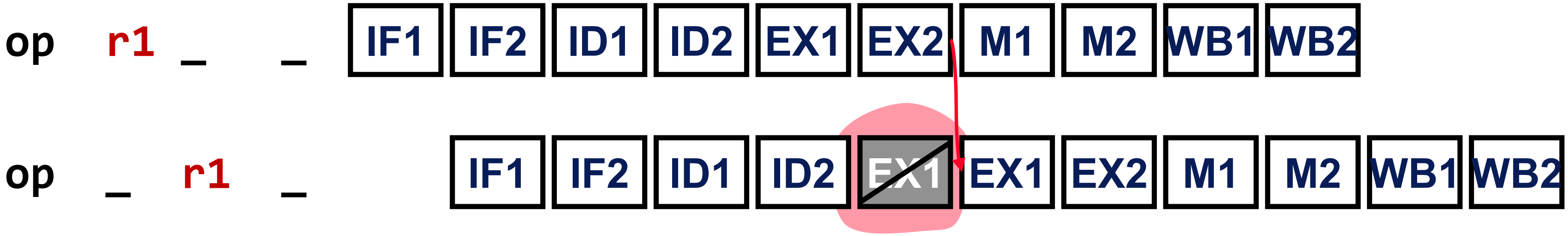
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11} t_{12}
$Inst_0$
$Inst_1$
$Inst_2$
$Inst_3$
$Inst_4$
IF
ID
EX
M
WB

This assumes higher clock frequency…

Recall: Is 5-stage pipeline sufficient?

- There are still plenty of combinational delay between regiters
- “**Superpipelining**” increases pipelining degree such that even intrinsic operations (e.g., ALU, RF read/write, memory access) require multiple stages

- Potential problem: “more data hazards”



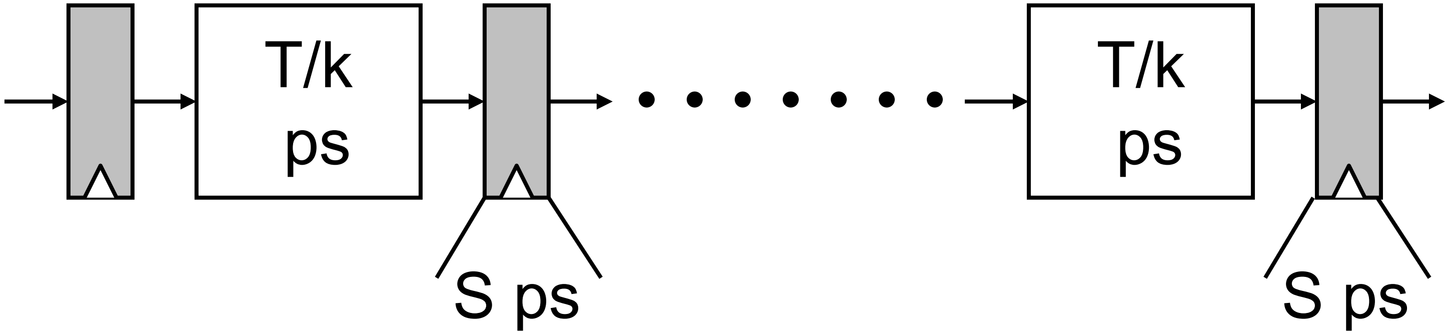
op r1 _ _ IF1 IF2 ID1 ID2 EX1 EX2 M1 M2 WB1 WB2

op _ r1 _ IF1 IF2 ID1 ID2 ~~EX1~~ EX1 EX2 M1 M2 WB1 WB2

Pipeline stall even w/ forwarding

Recall: But above all, you cannot pipeline till infinity!

- There are diminishing returns on clock speed



- Complicates hardware (there should be tons of wires for data forwarding)

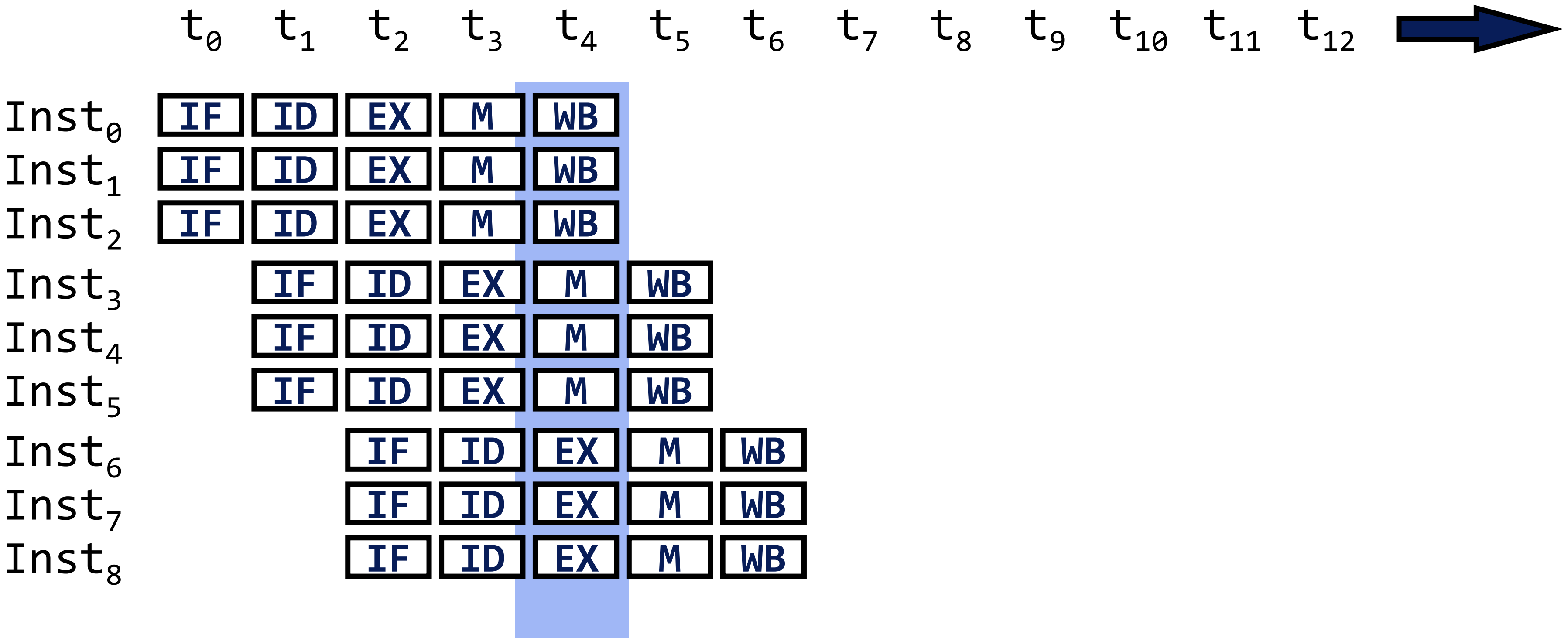
- You cannot always solve the dependency problem w/ forwarding!

We need to use superscalar!!!

Superscalar Machines

- Superscalar (+ pipelined) execution
  - **Operation latency =** 1 baseline cycle
  - **Peak IPC =** N per baseline cycle
  - **HW ILP =** # of instructions / # of cycles required = N

Peak throughpute
→ N배

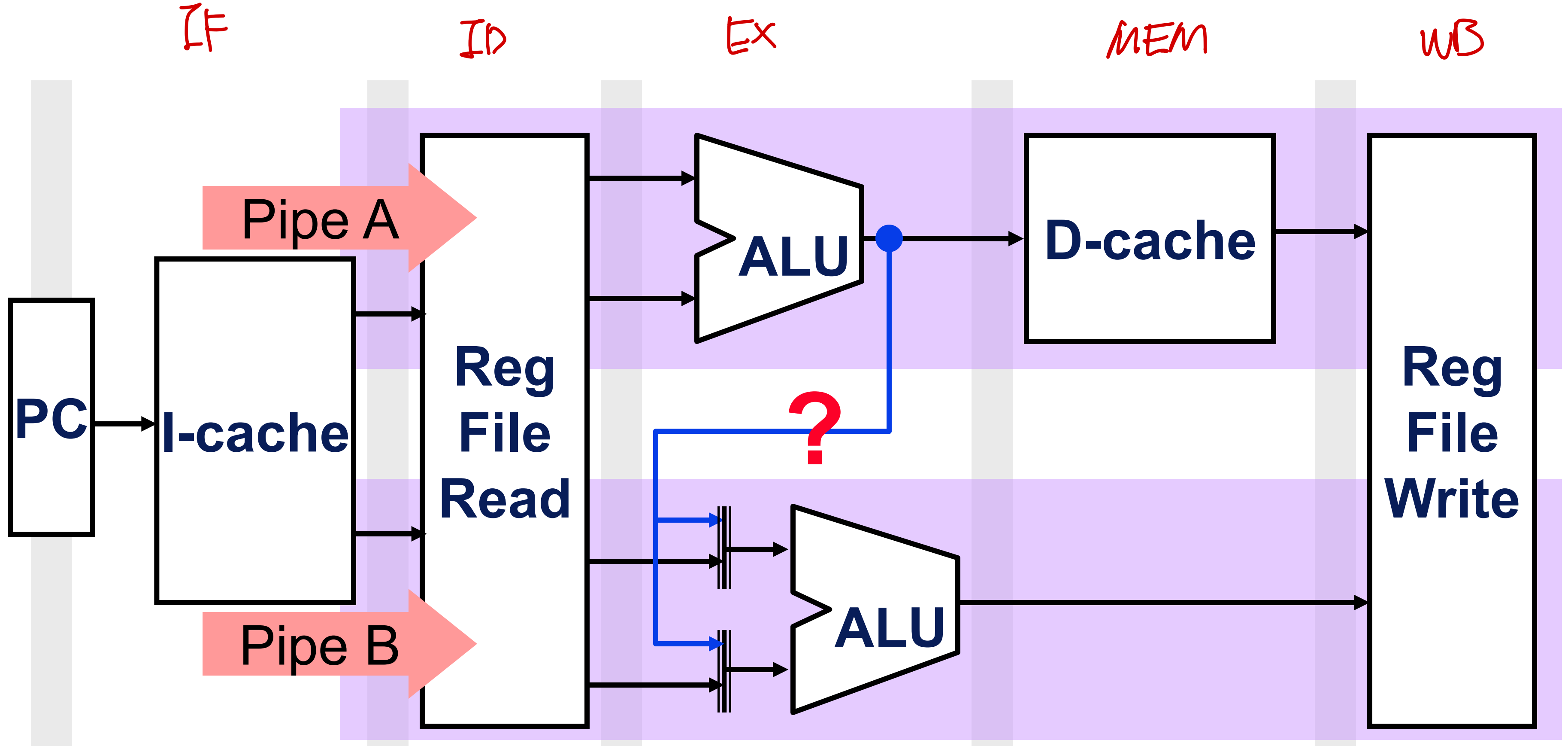


	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
Inst_0	IF	ID	EX	M	WB								
Inst_1	IF	ID	EX	M	WB								
Inst_2	IF	ID	EX	M	WB								
Inst_3		IF	ID	EX	M	WB							
Inst_4		IF	ID	EX	M	WB							
Inst_5		IF	ID	EX	M	WB							
Inst_6			IF	ID	EX	M	WB						
Inst_7			IF	ID	EX	M	WB						
Inst_8			IF	ID	EX	M	WB						

메모리 접근 제외 모두 2배

비용측면에서 2배 이상도 곤란함

Example 2-way superscalar datapath



2 X fetch bandwidth

2 X read ports

2 X Logic

Can't always double resources (high cost!)

2 X write ports

Superscalar vs. Superpipelined

	t_0	t_1	t_2	t_3	t_4	t_5
Inst_0	IF	ID	EX	M	WB	
Inst_1	IF	ID	EX	M	WB	
Inst_2	IF	ID	EX	M	WB	
Inst_3		IF	ID	EX	M	WB
Inst_4		IF	ID	EX	M	WB
Inst_5		IF	ID	EX	M	WB

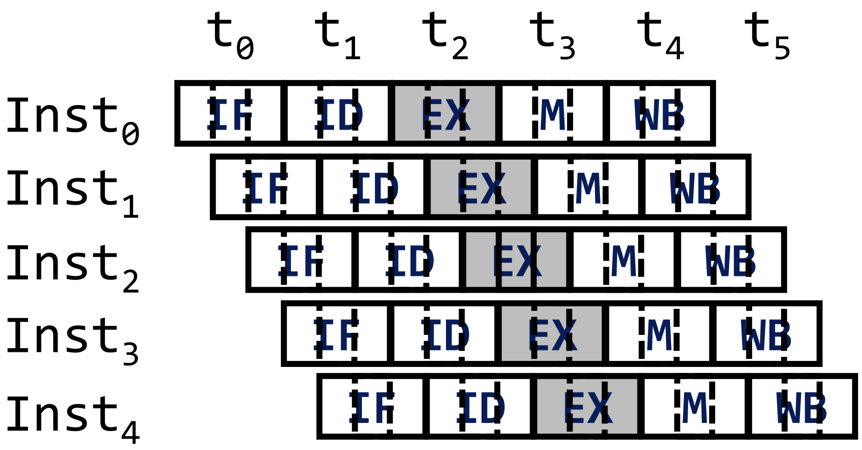
Superscalar Parallelism

Operation Latency: 1

Issuing Rate: **N**

Superscalar Degree: **N**

vs.



Superpipeline Parallelism

Operation Latency: 1

Issuing Rate: **M**

Superpipelined Degree: **M**

We need to exploit both methodologies (N and M) to maximize the performance benefits

How to maximize the ILP?
Instruction scheduling

- **Static scheduling (or VLIW)**
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards

동시에 실행할 수 있는 Independent한 Instruction 들을 Grouping 해줌

- **Dynamic scheduling**
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions (out-of-order execution)
  - CPU resolves hazards using advanced techniques at runtime

Compiler Cost : Static > Dynamic

요즘 사용

VLIW (Very Long Instruction Word)

- Compiler groups instructions into "issue packets"
  - It determines the group of instructions that can be issued on a single cycle

    하나의 Cycle내에서 Dependency, resource 확인
  - Determined by pipeline resources!

    Adder가 몇개인지 등

- You can make a very long instruction (specifies multiple concurrent operations)

  CPU 입장에서는 하나의 Instruction (packet)을 읽어오는데, 여러 개의 Instruction이 묶여있는 것

- Compiler must remove some/all hazards

  컴파일러 time에 모든 Data dependency를 없애줌 → 성능 ↓↓
  - Reorder instructions into issue packets
  - No dependencies within a packet
    - True or false dependencies (or both?) ← 둘 다 고려!
    - Pad with nop if necessary (cannot be parallelized)
  - Possibly some dependencies between packets

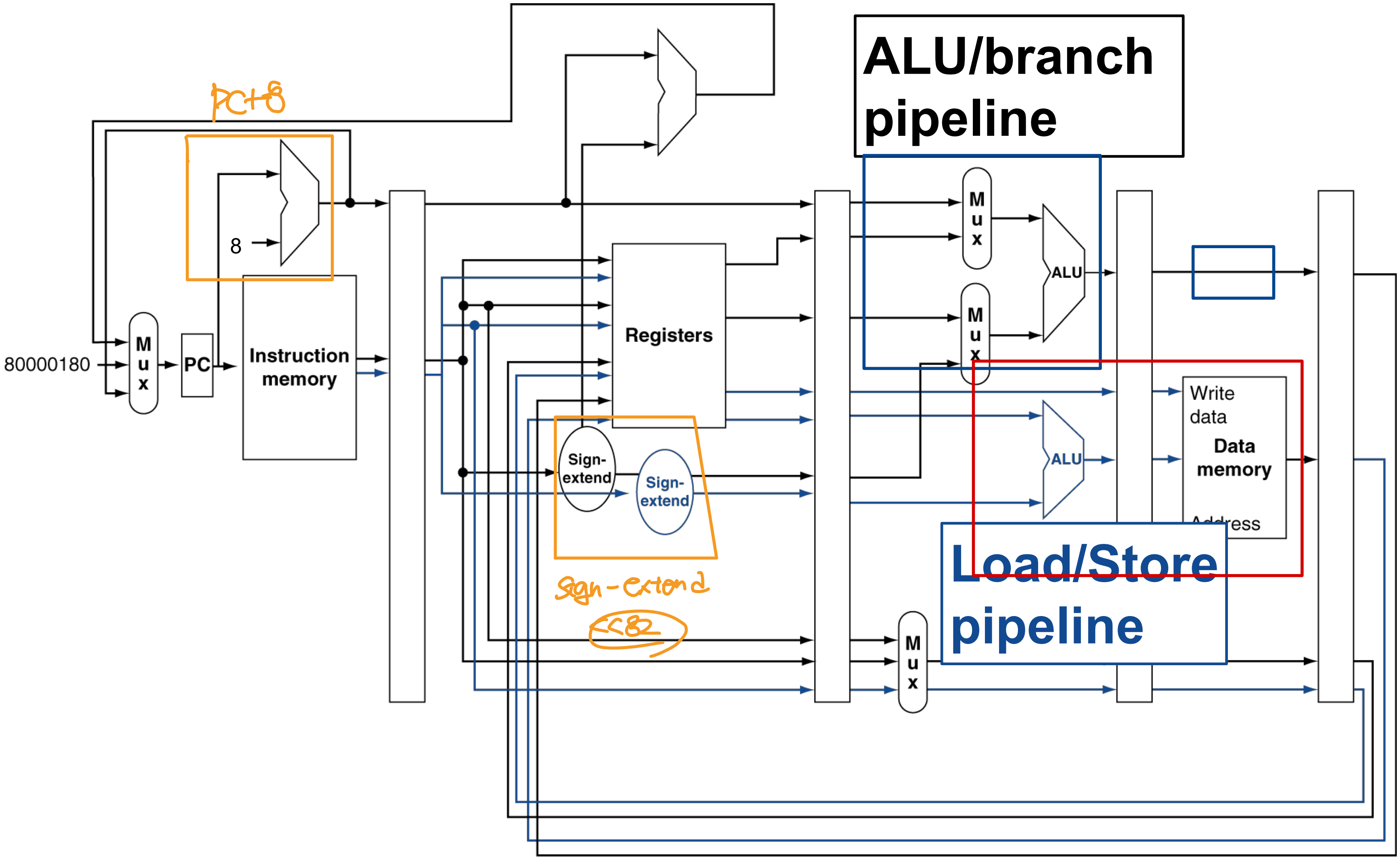
MIPS with static dual issue

- Two-issue packets! : 하나의 packet 안에 2개의 Instruction
  - One ALU/branch instruction ← MEM stage X
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop ←포기

Pc 다음에 PC+8로 Fetch

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	Bubble ~~MEM~~	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	Bubble ~~MEM~~	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	Bubble ~~MEM~~	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

MIPS with static dual issue



ALU/branch pipeline

Load/Store pipeline

Hazards in the dual-issue MIPS

- More instructions are executed in parallel
- **EX data hazard**
  - Can’t use ALU result in load/store in same packet

```
Slot 0 { add  $t0, $s0, $s1
Slot 1 { load $s2, 0($t0)
```

Split into two packets, effectively a stall

- **Load-use hazard**
  - Still one cycle use latency

```
Slot 0 { load $t0, 0($s0)
Slot 1 { add  $t2, $t0, $s1
```

1 cycle stall

Split into two packets, effectively a stall

Hazards in the dual-issue MIPS

- It also suffers from false dependencies
- **Write after read hazard**
  - You cannot place two instruction with WAR hazard

    ```
    add  $t0, $s0, $s1
    load $s0, 0($t1)
    ```

    Split into two packets, effectively a stall
- **Write after write hazard**
  - The two packed instructions cannot write to the same register

    ```
    load $t0, 0($s0)
    add  $t0, $t1, $s1
    ```

    Split into two packets, effectively a stall

Scheduling example

- Schedule instructions in the loop for dual-issue MIPS!

```
addi $s1, $zero, 400                # $s1 = 400
Loop:
      lw   $t0, 0($s1)              # $t0=array element
      addu $t0, $t0, $s2            # add scalar in $s2
      sw   $t0, 0($s1)              # store result
      addi $s1, $s1,-4              # decrement pointer
      bne  $s1, $zero, Loop         # branch $s1!=0
```

100번 loop

	ALU/branch	Load/store	cycle
Loop:	nop *Delay Slot*	lw $t0, 0($s1)	1
	addi $s1, $s1,-4	nop	2
	addu $t0, $t0, $s2	nop	3
	bne $s1, $zero, Loop	sw $t0, 4($s1)	4

IPC = 5 / 4 = 1.25 (2-way superscalar)

Loop unrolling

4번 Unrolling
→ 100번 도는 loop를
25번 돌도록

- Replicate loop body to expose more parallelism + reduce loop-control overhead

```
# Unroll loop by 4
addi $s1, $zero, 400                        # $s1 = 400
Loop:
       lw   $t0, 0($s1)
       addu $t0, $t0, $s2
       sw   $t0, 0($s1)
       lw   $t0, -4($s1)
       addu $t0, $t0, $s2
       sw   $t0, -4($s1)
       lw   $t0, -8($s1)
       addu $t0, $t0, $s2
       sw   $t0, -8($s1)
       lw   $t0, -12($s1)
       addu $t0, $t0, $s2
       sw   $t0, -12($s1)
       addi $s1, $s1, -16
       bne  $s1, $zero, Loop # branch $s1!=0
```

← 원래는 -4, 4 Unroll이니까 -16

Decrement + Branch once per four loops

Loop unrolling & Scheduling

명령어 개수↓, cycle 개수는 유지 → Low IPC 개선

한 Loop에서 명령어 5개, 필요한 cycle 4개 $\xrightarrow{\text{Unrolling}}$ 4개 loop에 대해 명령어 14개, 그러나 cycle은 16개

	ALU/branch	Load/store	cycle
Loop:	nop	lw $t0, 0($s1)	1
	nop (stall)	nop	2
	addu $t0, $t0, $s2	nop	3
	nop	sw $t0, 0($s1)	4
	nop	lw $t0, -4($s1)	5
	nop (stall)	nop	6
	addu $t0, $t0, $s2	nop	7
	nop	sw $t0, -4($s1)	8
	nop	lw $t0, -8($s1)	9
	nop (stall)	nop	10
	addu $t0, $t0, $s2	nop	11
	nop	sw $t0, -8($s1)	12
	nop	lw $t0, -12($s1)	13
	addi $s1, $s1, -16	nop	14
	addu $t0, $t0, $s2	nop	15
	bne $s1, $zero, Loop	sw $t0, 4($s1)	16

WAW

WAW

WAR

Loop unrolling & Scheduling

	ALU/branch	Load/store	cycle
Loop:	nop	lw $t0, 0($s1)	1
	nop (stall)	nop	2
	addu $t0, $t0, $s2	nop	3
	nop	sw $t0, 0($s1)	4
	nop	lw	5
	nop (stall)		6
			11
		sw $t0, -8($s1)	12
		lw $t0, -12($s1)	13
	addi $s1, $s1, -16	nop	14
	addu $t0, $t0, $s2	nop	15
	bne $s1, $zero, Loop	sw $t0, 4($s1)	16

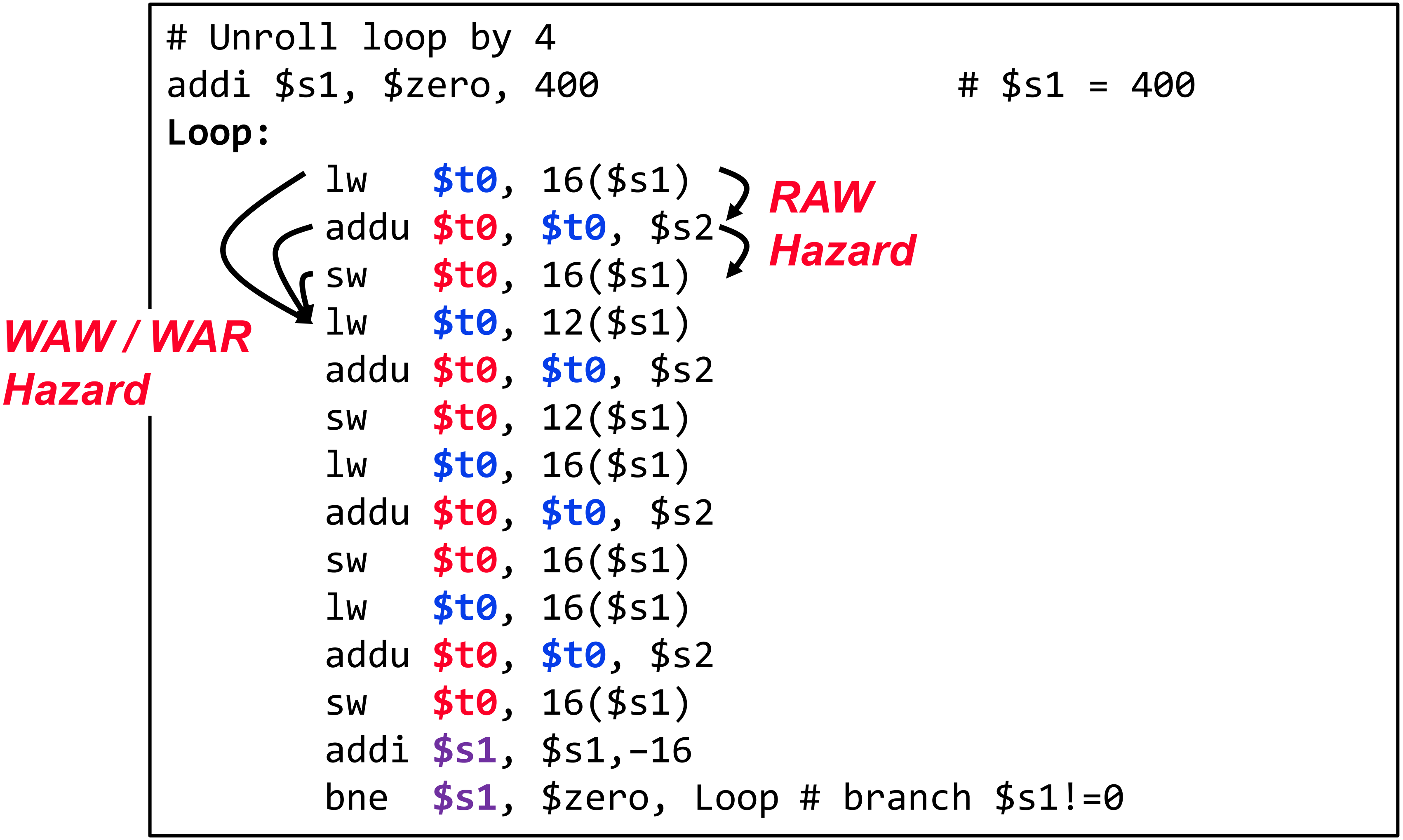
16 cycles per 4 loops (no cycle reduction) even though the # of instructions are reduced

Because … consider IPC = (14 / 16) / < 1.25

➔We do not benefit from the improved parallelization opportunities

Why low IPC?

- There is an **extremely large # of dependencies**



```
# Unroll loop by 4
addi $s1, $zero, 400                    # $s1 = 400
Loop:
       lw   $t0, 16($s1)
       addu $t0, $t0, $s2
       sw   $t0, 16($s1)
       lw   $t0, 12($s1)
       addu $t0, $t0, $s2
       sw   $t0, 12($s1)
       lw   $t0, 16($s1)
       addu $t0, $t0, $s2
       sw   $t0, 16($s1)
       lw   $t0, 16($s1)
       addu $t0, $t0, $s2
       sw   $t0, 16($s1)
       addi $s1, $s1,-16
       bne  $s1, $zero, Loop # branch $s1!=0
```

RAW Hazard

WAW / WAR Hazard

SW-based Register renaming

- Compiler renames registers to remove false dependencies

```
# Unroll loop by 4
addi $s1, $zero, 400                 # $s1 = 400
Loop:
        lw   $t0, 16($s1)
        addu $t0, $t0, $s2
        sw   $t0, 16($s1)
        lw   $t1, 12($s1)
        addu $t1, $t1, $s2
        sw   $t1, 12($s1)
        lw   $t2, 16($s1)
        addu $t2, $t2, $s2
        sw   $t2, 16($s1)
        lw   $t3, 16($s1)
        addu $t3, $t3, $s2
        sw   $t3, 16($s1)
        addi $s1, $s1,-16
        bne  $s1, $zero, Loop # branch $s1!=0
```

Scheduling after renaming

	ALU/branch	Load/store	cycle
Loop:	addi $s1, $s1,-16	nop	1
	nop	lw $t0, 16($s1)	2
	nop	lw $t1, 12($s1) Stall 대신 lw 다른 reg	3
	addu $t0, $t0, $s2	lw $t2, 8($s1)	4
	addu $t1, $t1, $s2	lw $t3, 4($s1)	5
	addu $t2, $t2, $s2	sw $t0, 16($s1)	6
	addu $t3, $t3, $s2	sw $t1, 12($s1)	7
	nop	sw $t2, 8($s1)	8
	bne $s1, $zero, Loop	sw $t3, 4($s1)	9

Scheduling after renaming

	ALU/branch	Load/store	cycle
Loop:	addi $s1, $s1,-16	nop	1
	nop	lw $t0, 16($s1)	2
	nop	lw $t1, 12($s1)	3
	addu $t0, $t0, $s2		4
	add		5
			9

9 cycles per 4 loops (7 cycle reduction)

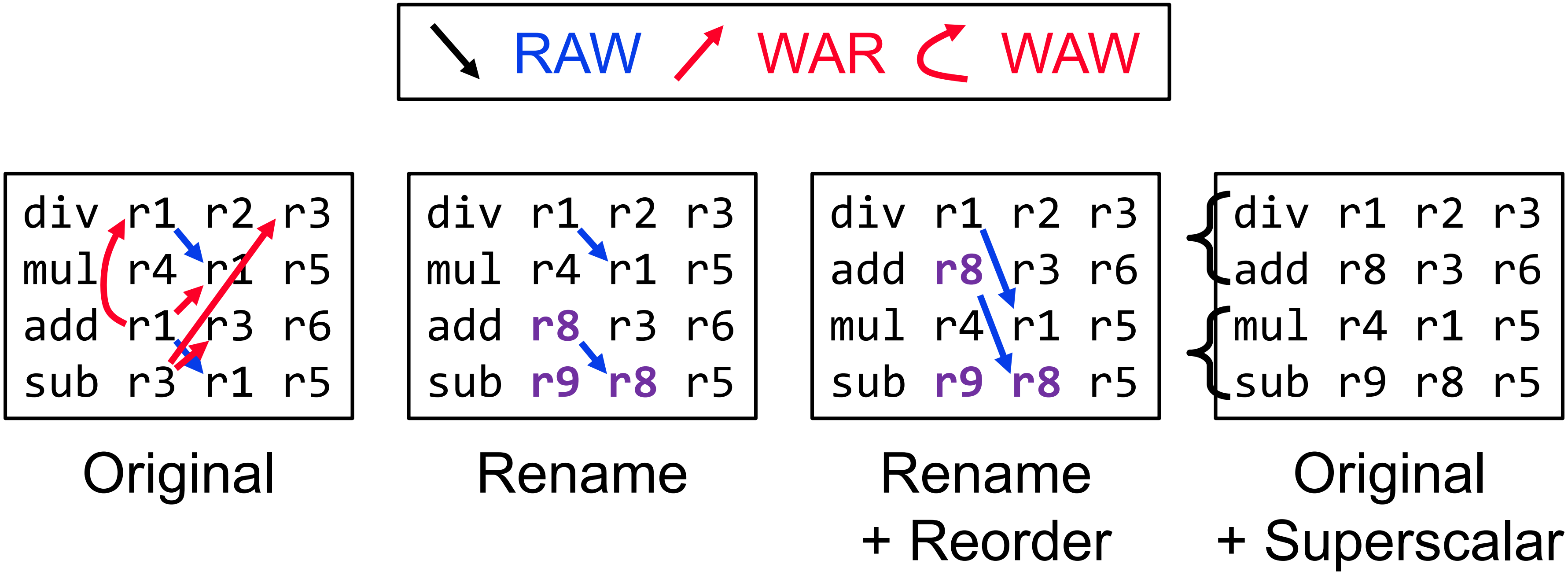
IPC가 증가하는 효과!

Consider IPC = 1.555 > 1.25 → $\frac{20}{16}=1.25$

➔ Benefits from the improved parallelization opportunities

Sum up …
Renaming & scheduling increases ILP!!

- We can execute previously dependent (false dependency) instructions in parallel



```
div r1 r2 r3
mul r4 r1 r5
add r1 r3 r6
sub r3 r1 r5
```

Original

```
div r1 r2 r3
mul r4 r1 r5
add r8 r3 r6
sub r9 r8 r5
```

Rename

```
div r1 r2 r3
add r8 r3 r6
mul r4 r1 r5
sub r9 r8 r5
```

Rename
+ Reorder

```
div r1 r2 r3
add r8 r3 r6
mul r4 r1 r5
sub r9 r8 r5
```

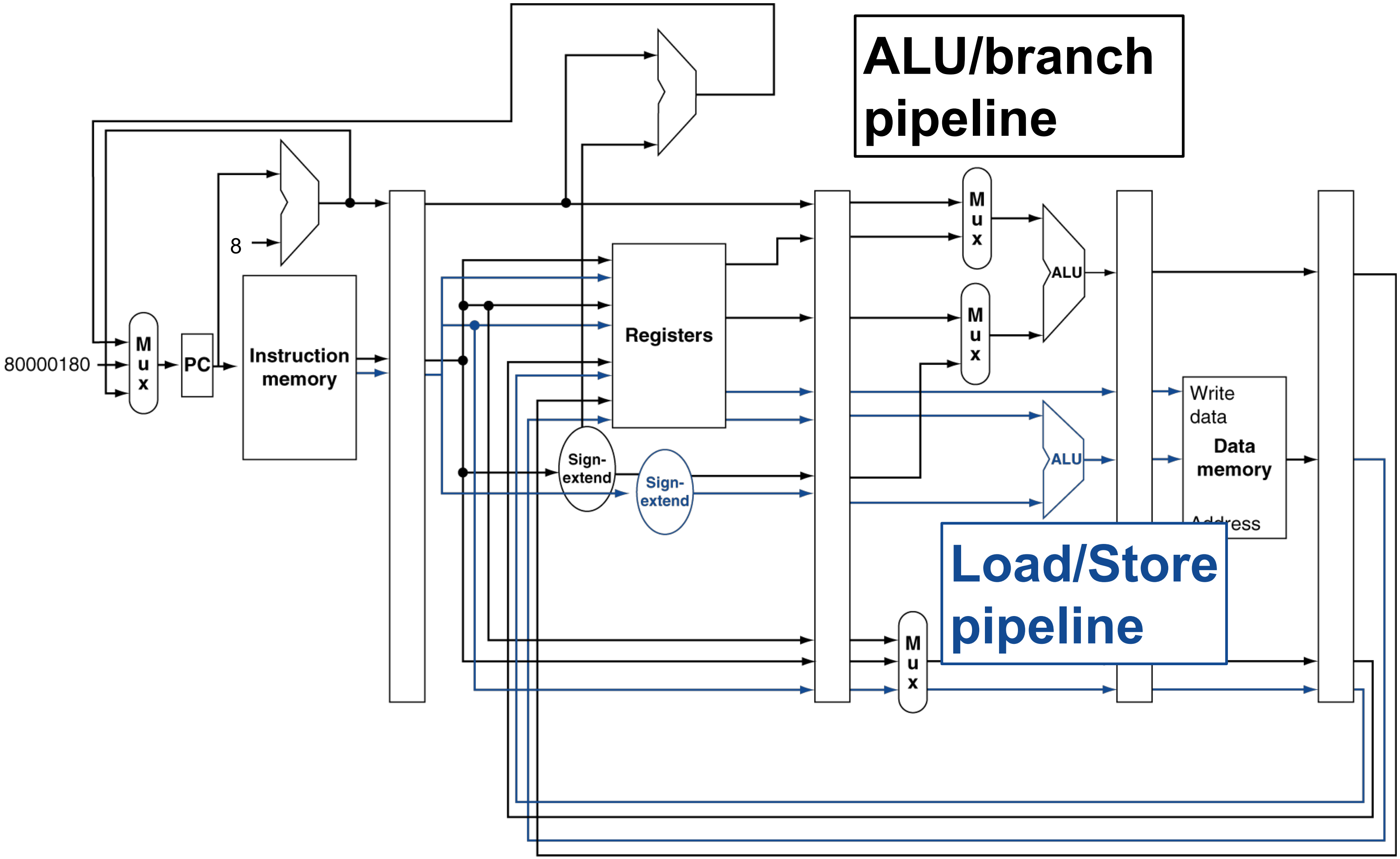
Original
+ Superscalar

MIPS with static dual issue

- Two-issue packets!
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

MIPS with static dual issue



ALU/branch pipeline

Load/Store pipeline

© Lee 2024 -- Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch, Mutlu, Kim

How to maximize the ILP? Instruction scheduling

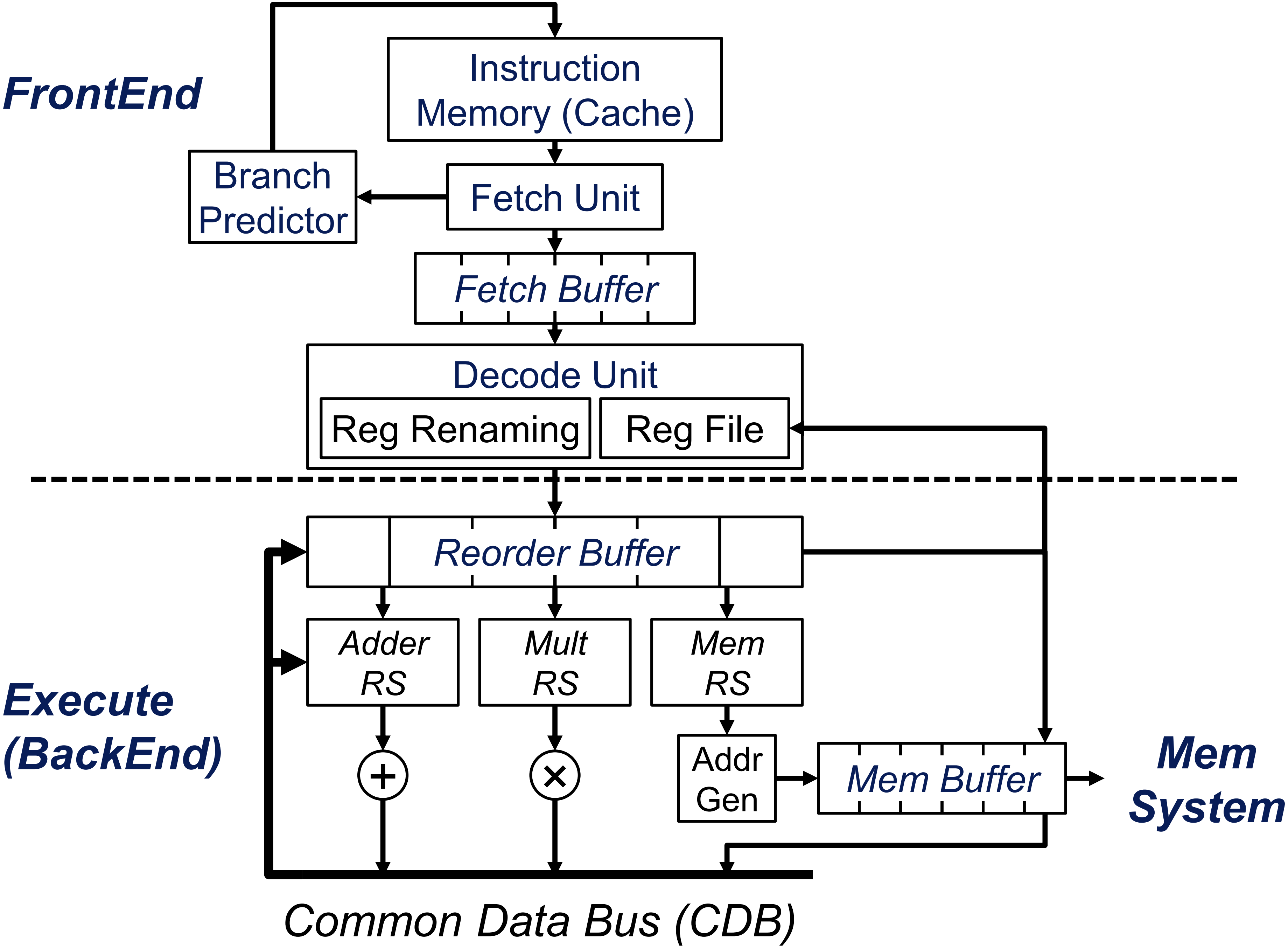
- **Static scheduling (or VLIW)**
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards

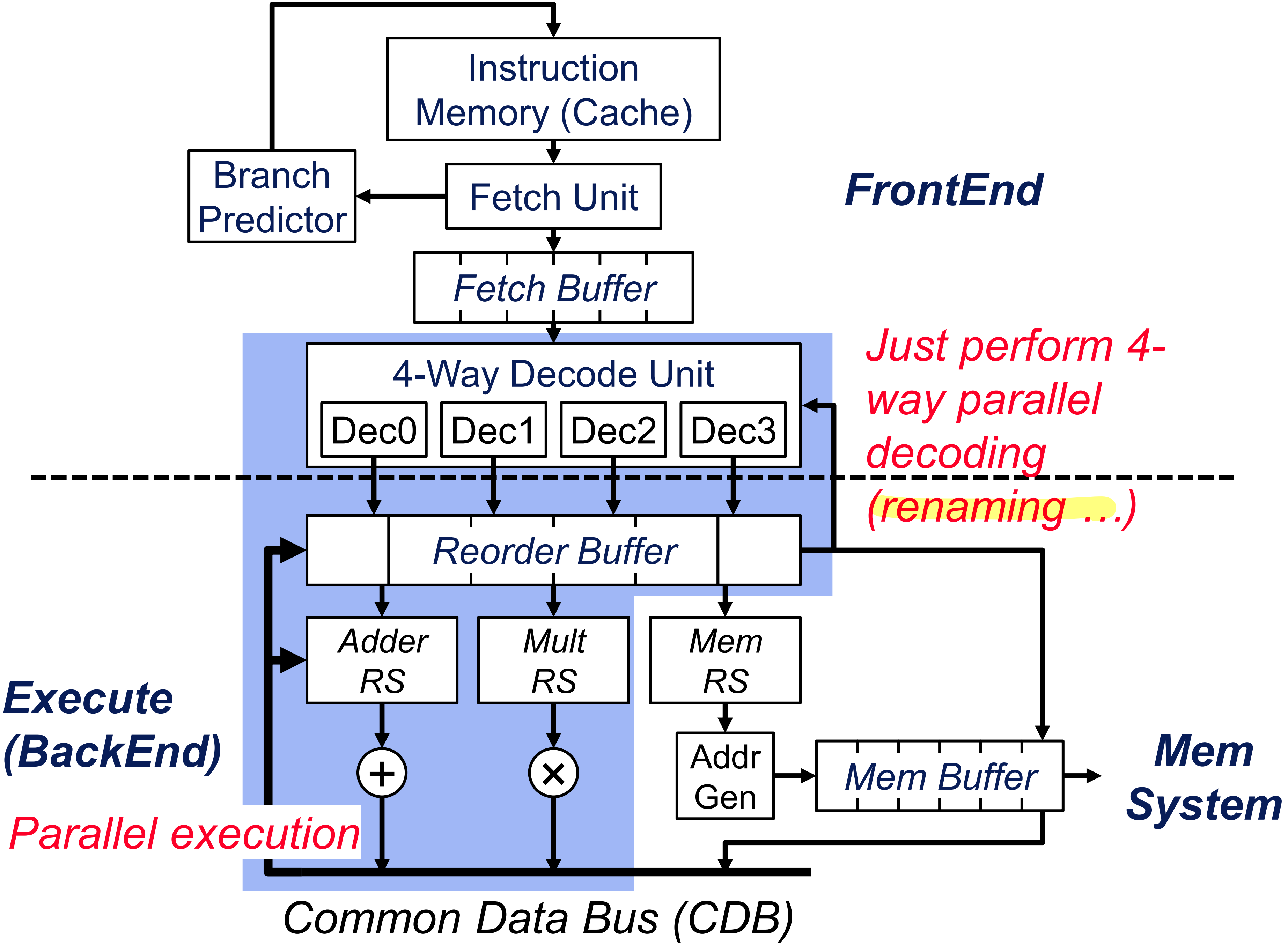
- **Dynamic scheduling**
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions (out-of-order execution)
  - CPU resolves hazards using advanced techniques at runtime

Superscalar Processing

- Parallel N-instruction fetch
- Parallel N-instruction decode
- Parallel N-instruction renaming → (detect dependencies in parallel)
  - Simultaneously allocate ROB, RS …

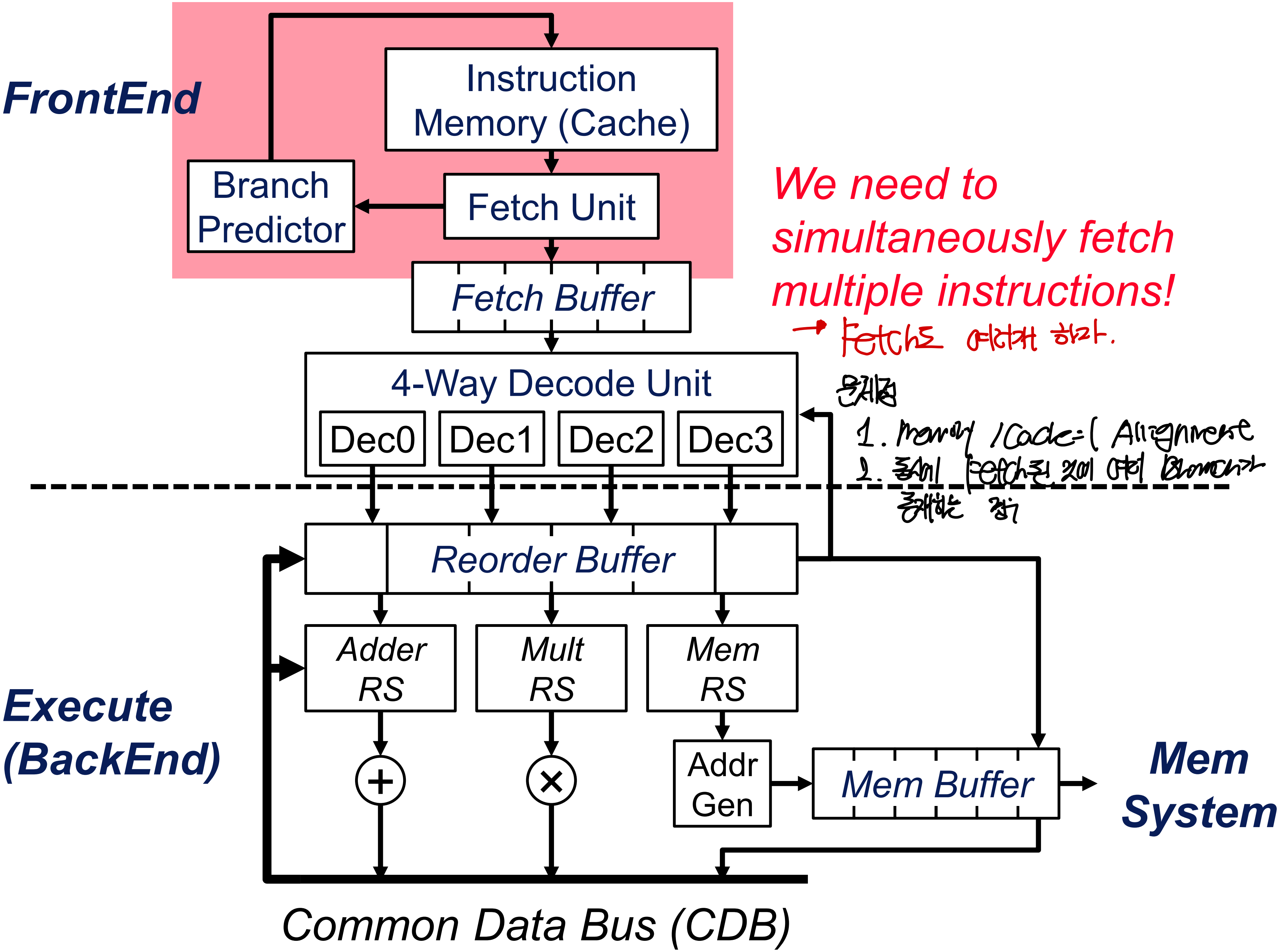
    → 동시에 N개의 명령어들 입력될 수 있도록~
- Dispatch N instructions in parallel (different execution units)
- Commit N instructions in parallel (N write ports RF)
- …


FrontEnd
Instruction Memory (Cache)
Branch Predictor
Fetch Unit
Fetch Buffer
Decode Unit
Reg Renaming
Reg File
Reorder Buffer
Adder RS
Mult RS
Mem RS
+
×
Addr Gen
Mem Buffer
Execute (BackEnd)
Mem System
Common Data Bus (CDB)



Just perform 4-way parallel decoding (renaming …)

Parallel execution



We need to simultaneously fetch multiple instructions!

→ Fetch도 여러개 하라.

문제점

1. Memory / Cache의 Allignment

2. 동시에 Fetch된 것에 여러 Branch가 존재하는 경우

© Lee 2024 -- Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch, Mutlu, Kim

Fetching Multiple Instructions

- **Problem #1:** alignment problem with memory (cache)
    - You are not able to read instructions at different cache lines in parallel

- **Problem #2:** branches in-between N instruction sequence?
    - Should predict multiple branches in parallel
    - This incurs additional alignment issues

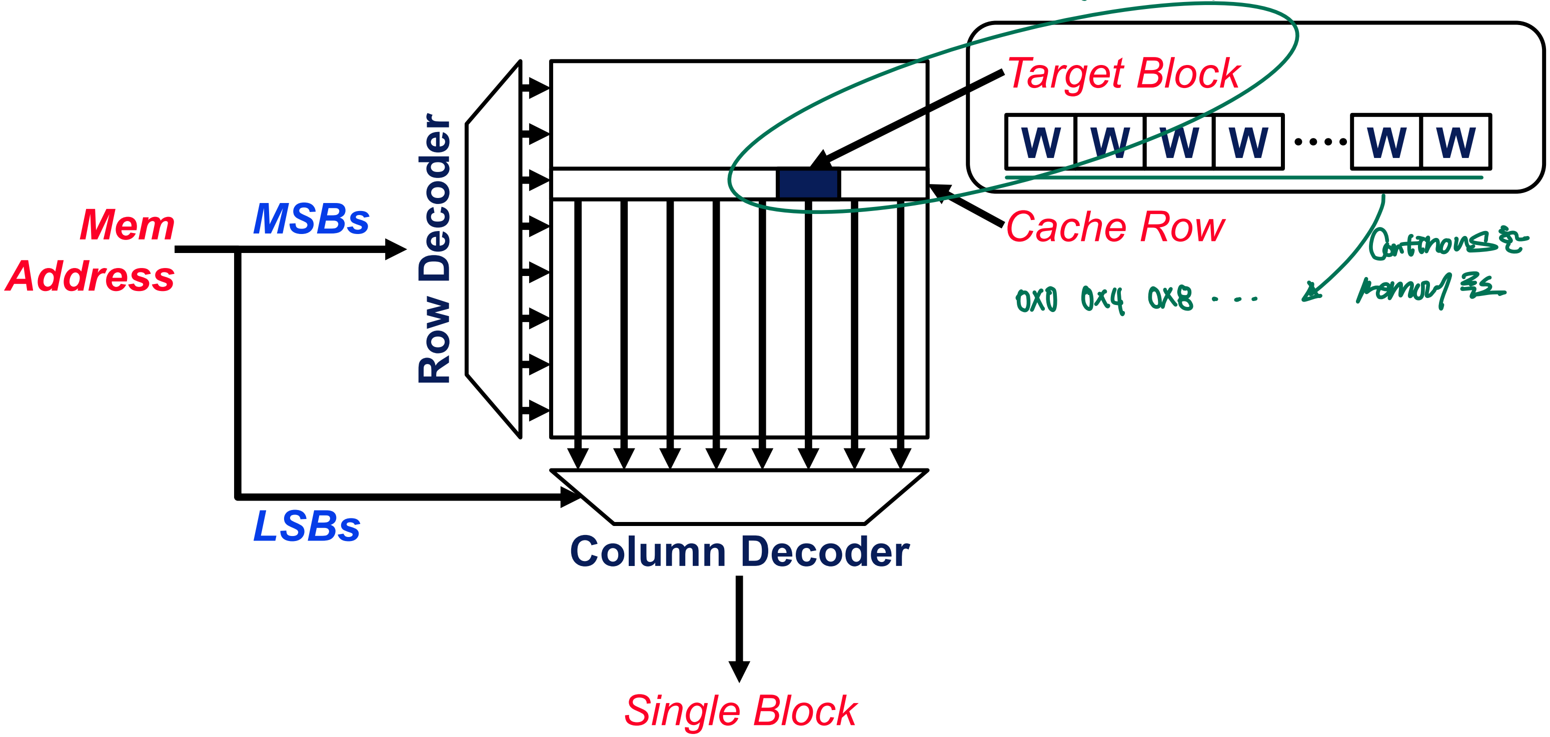
Preview on the Cache

- Cache is designed to read multiple words within a single cache block (typically 4~8 words)

Single Cycle에 Cache에서 읽을 수 있는 단위: Block

- The CPU can access 4 ~ 8 instructions in parallel (only if they are within the same block)

Single Block 안에 4~8개의 Word

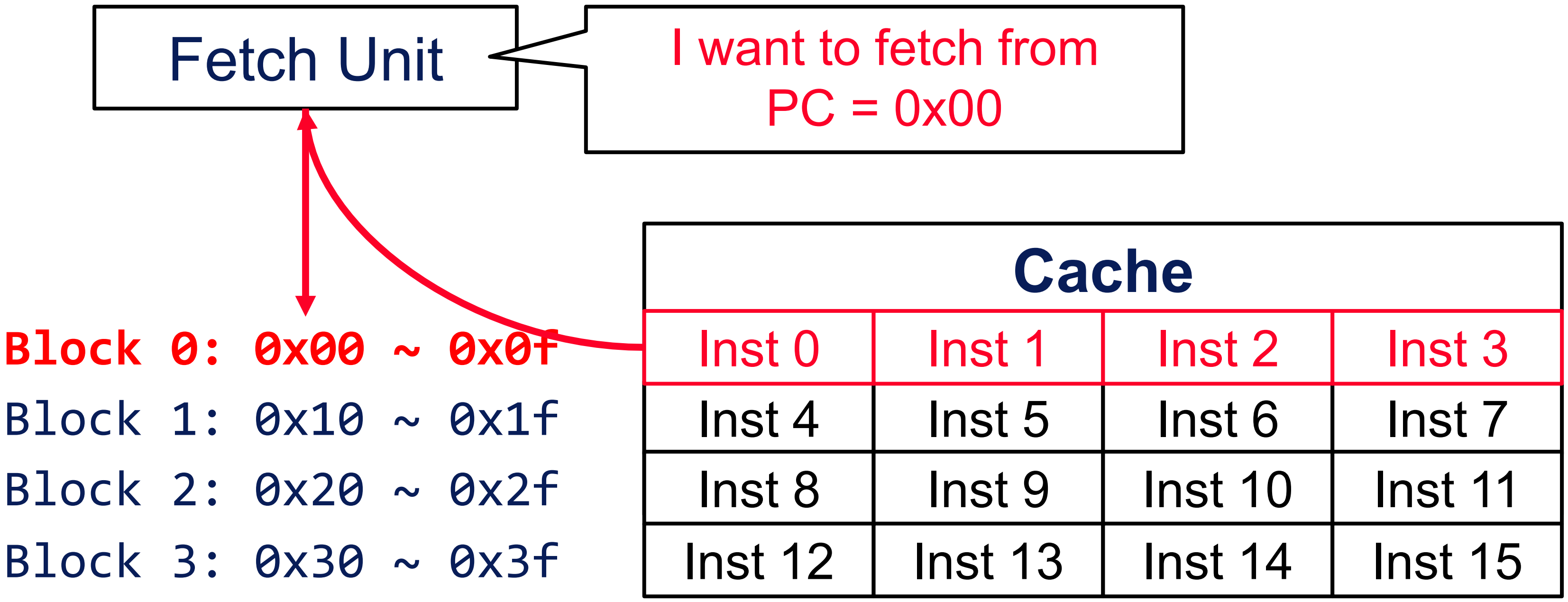


0x0 0x4 0x8 . . .

Continous한 memory 주소

Cache misalignment problem

- Consider a cache with the block size of four words to support four-way superscalar



Fetch Unit

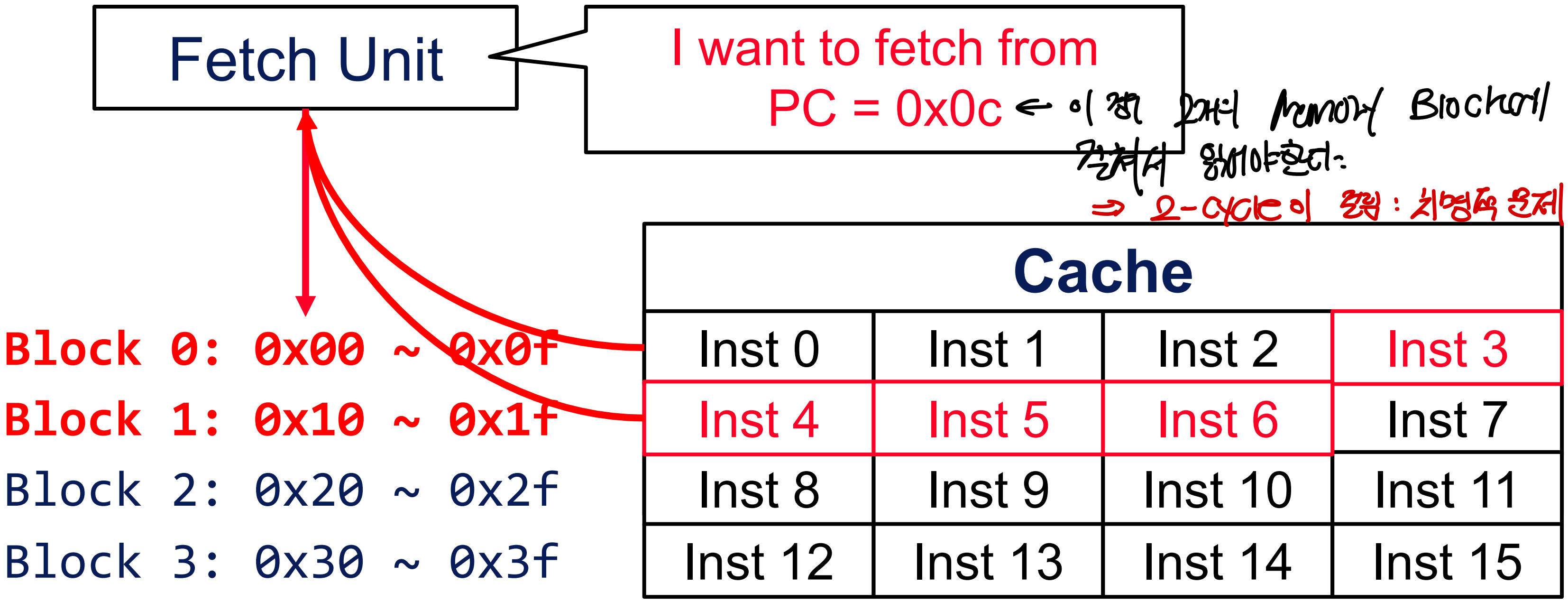
I want to fetch from PC = 0x00

```
Block 0: 0x00 ~ 0x0f
Block 1: 0x10 ~ 0x1f
Block 2: 0x20 ~ 0x2f
Block 3: 0x30 ~ 0x3f
```

Cache			
Inst 0	Inst 1	Inst 2	Inst 3
Inst 4	Inst 5	Inst 6	Inst 7
Inst 8	Inst 9	Inst 10	Inst 11
Inst 12	Inst 13	Inst 14	Inst 15

Cache misalignment problem

- Consider a cache with the block size of four words to support four-way superscalar



Fetch Unit

I want to fetch from PC = 0x0c ← 이 경우 2개의 Memory Block에 걸쳐서 읽어야한다.

⇒ 2-cycle이 걸림 : 치명적 문제

Block 0: 0x00 ~ 0x0f

Block 1: 0x10 ~ 0x1f

Block 2: 0x20 ~ 0x2f

Block 3: 0x30 ~ 0x3f

Cache			
Inst 0	Inst 1	Inst 2	Inst 3
Inst 4	Inst 5	Inst 6	Inst 7
Inst 8	Inst 9	Inst 10	Inst 11
Inst 12	Inst 13	Inst 14	Inst 15

There is one instruction @ Block 0
While others are @ Block 1

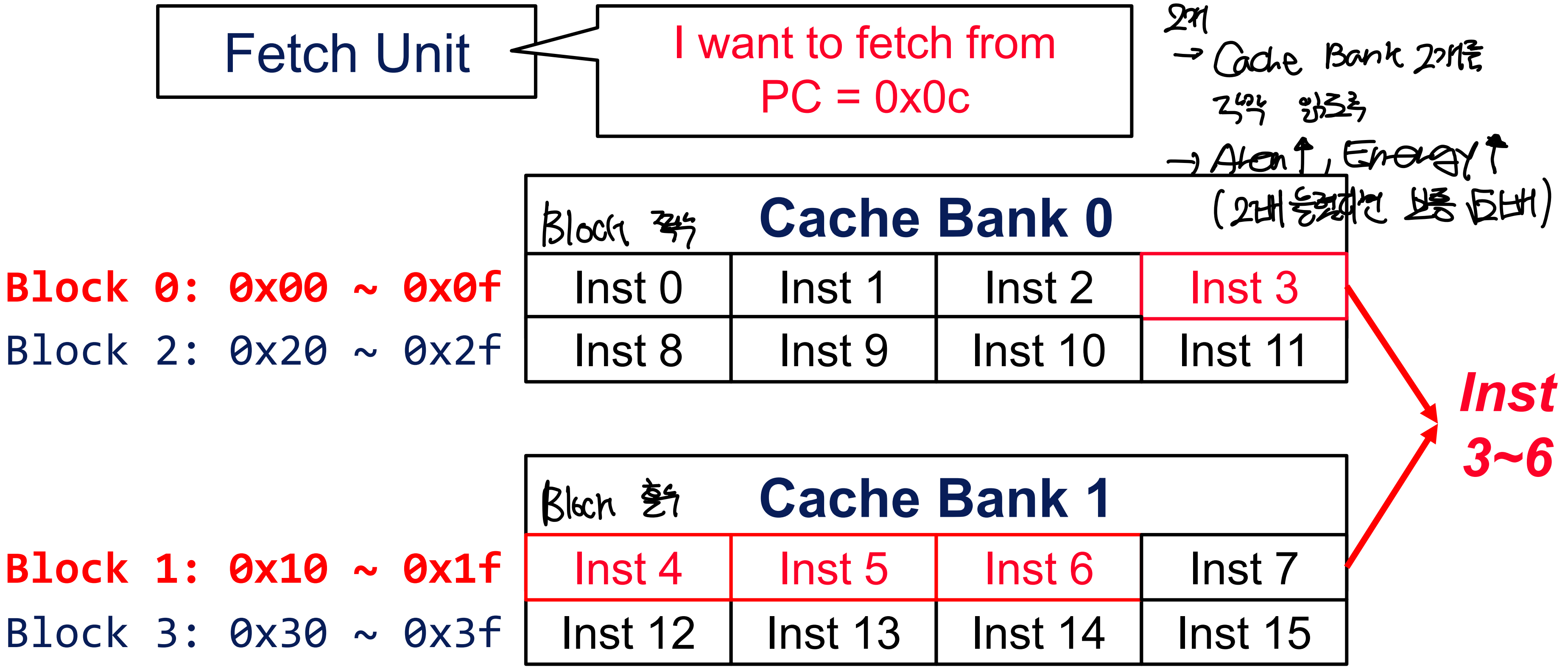
➔ ***Fetch becomes the bottleneck***

Split line fetching

가장 쉬운 Solution

- You can divide a cache to keep even blocks and odd blocks separately ➔ You can access the two blocks in parallel!

Block 읽어오는 장치를 2개
→ Cache Bank 2개를 각각 읽도록
→ Area↑, Energy↑
(2배 늘렸지만 보통 1.5배)



Fetch Unit

I want to fetch from PC = 0x0c

Block 0: 0x00 ~ 0x0f
Block 2: 0x20 ~ 0x2f

Block 짝수 Cache Bank 0			
Inst 0	Inst 1	Inst 2	Inst 3
Inst 8	Inst 9	Inst 10	Inst 11

Block 1: 0x10 ~ 0x1f
Block 3: 0x30 ~ 0x3f

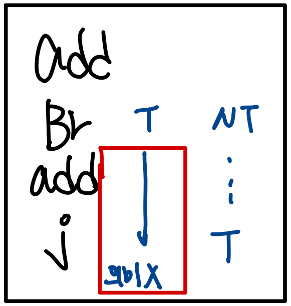
Block 홀수 Cache Bank 1			
Inst 4	Inst 5	Inst 6	Inst 7
Inst 12	Inst 13	Inst 14	Inst 15

Inst 3~6

Fetching Multiple Instructions

- **Problem #1:** alignment problem with memory (cache)
  - You are not able to read instructions at different cache lines in parallel

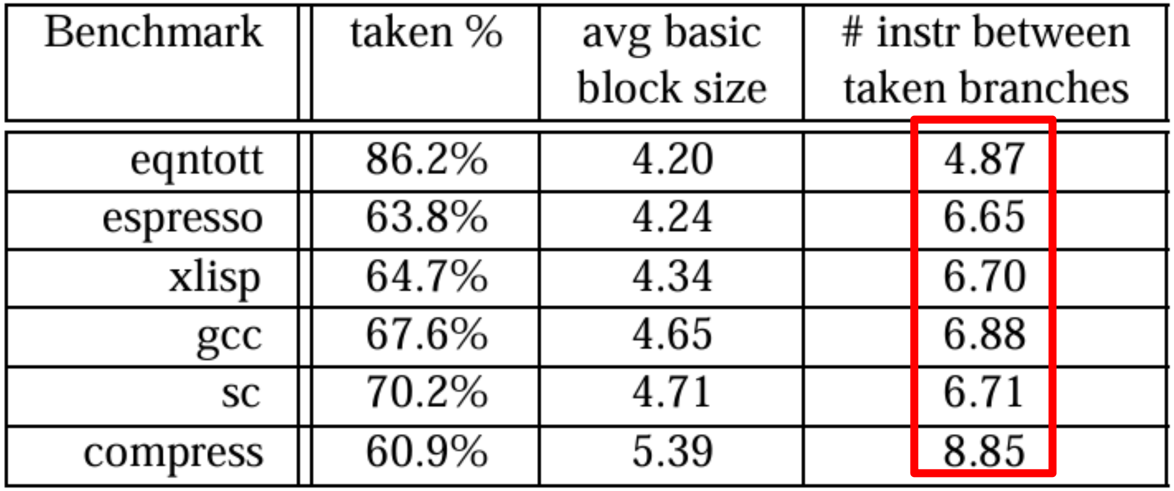
- **Problem #2:** branches in-between N instruction sequence? 4개 Instruction 중 여러 개의 Branch가 있다면
  - Should predict multiple branches in parallel
  - This incurs additional alignment issues



1. 여러 개의 Branch를 Parallel하게 예측할 수 있어야 한다.
2. Branch ↑ → Memory Allignment 문제 ↑

Increasing Branch Throughput

- There are a lot of branch instructions (15 ~ 20% of the instructions are branches)
- It is likely that there can be multiple branch instructions if you fetch N instructions (for superscalar)



Benchmark	taken %	avg basic block size	# instr between taken branches
eqntott	86.2%	4.20	4.87
espresso	63.8%	4.24	6.65
xlisp	64.7%	4.34	6.70
gcc	67.6%	4.65	6.88
sc	70.2%	4.71	6.71
compress	60.9%	5.39	8.85

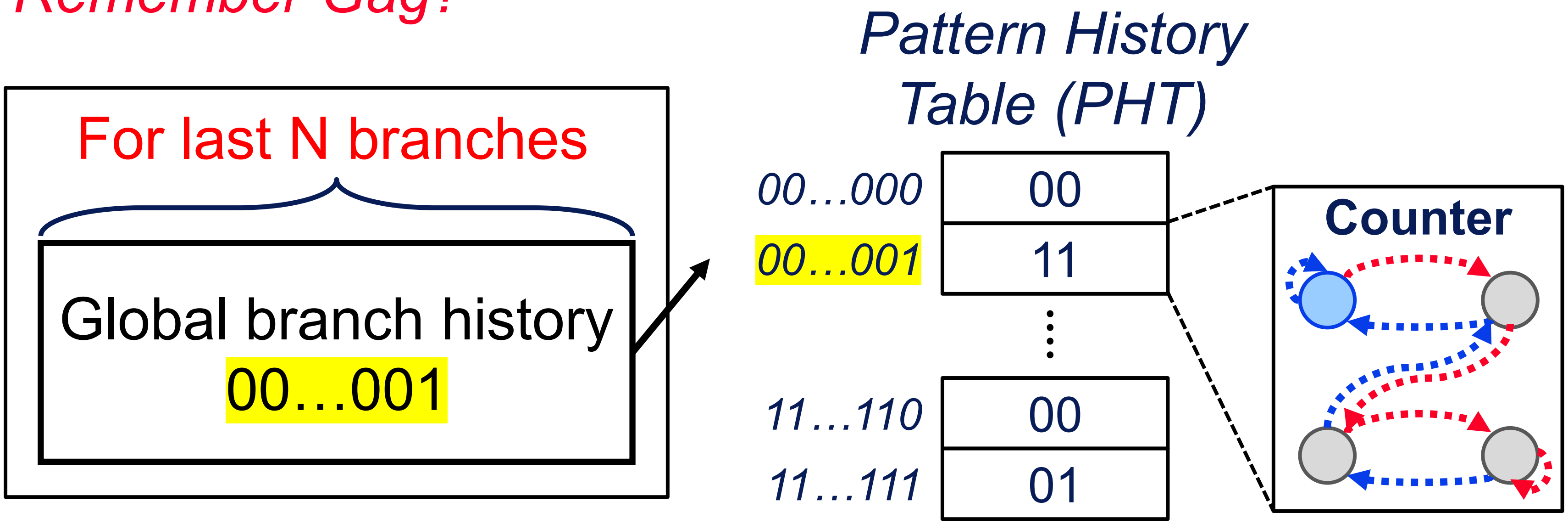
There is a branch every 4 ~ 8 instructions

Increasing Branch Throughput [ICS’93]

- Let’s say that there are an average of four instructions per basic block
  - ➔ Then, the maximum superscalar width would be four

- Extend Gag to predict multiple branches (MGag)

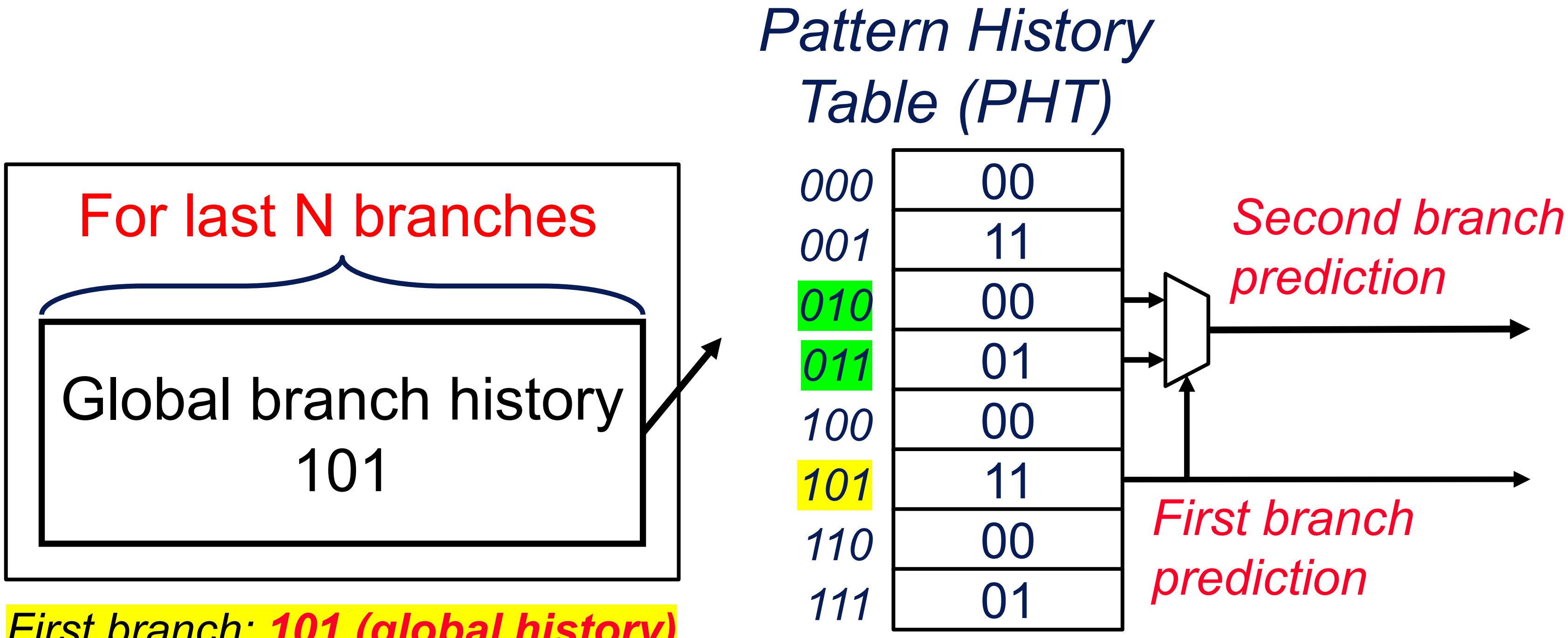
Remember Gag?



Increasing Branch Throughput [ICS'93]

- Let's say that there are an average of four instructions per basic block
  - ➔ Then, the maximum superscalar width would be four

- Extend Gag to predict multiple branches (MGag)



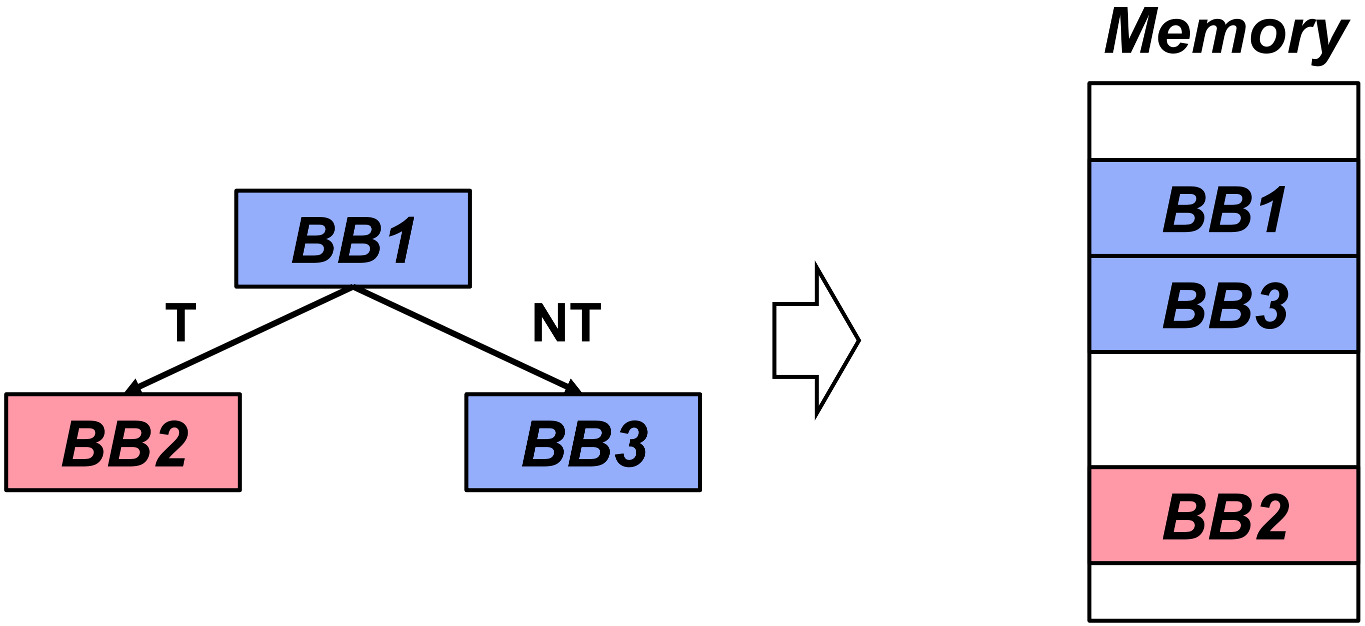
*First branch: **101 (global history)***

*Second branch: **01X (global history)***

© Lee 2024 -- Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch, Mutlu, Kim

Branch-induced fragmentation

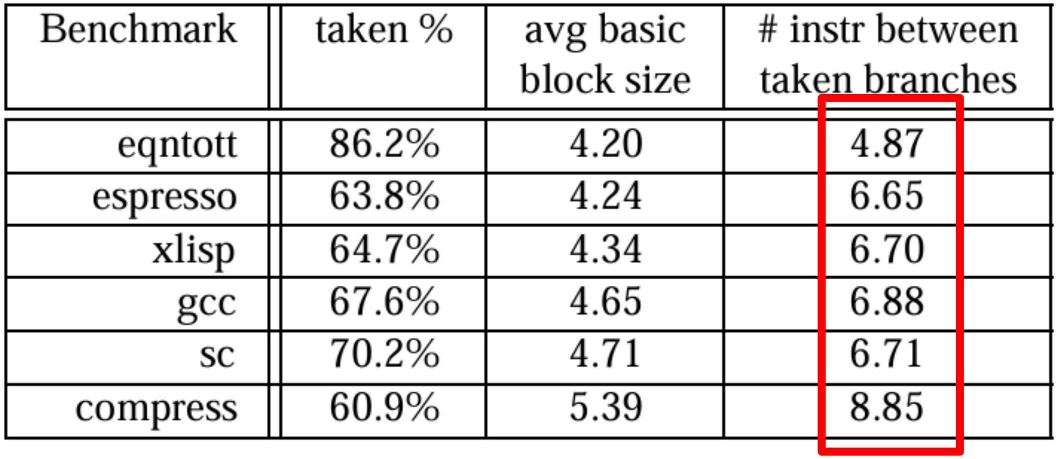
- Cache is good at accessing **contiguous data** (a single block contains multiple contiguous words)
- Taken instructions are placed in a **non-contiguous** region
- However, there are tons of taken branches in between (around 10~20%)



© Lee 2024 -- Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch, Mutlu, Kim

Branch-induced fragmentation

- Cache is good at accessing **contiguous data** (a single block contains multiple contiguous words)
- Taken instructions are placed in a **non-contiguous** region
- However, there are tons of taken branches in between (around 10~20%)

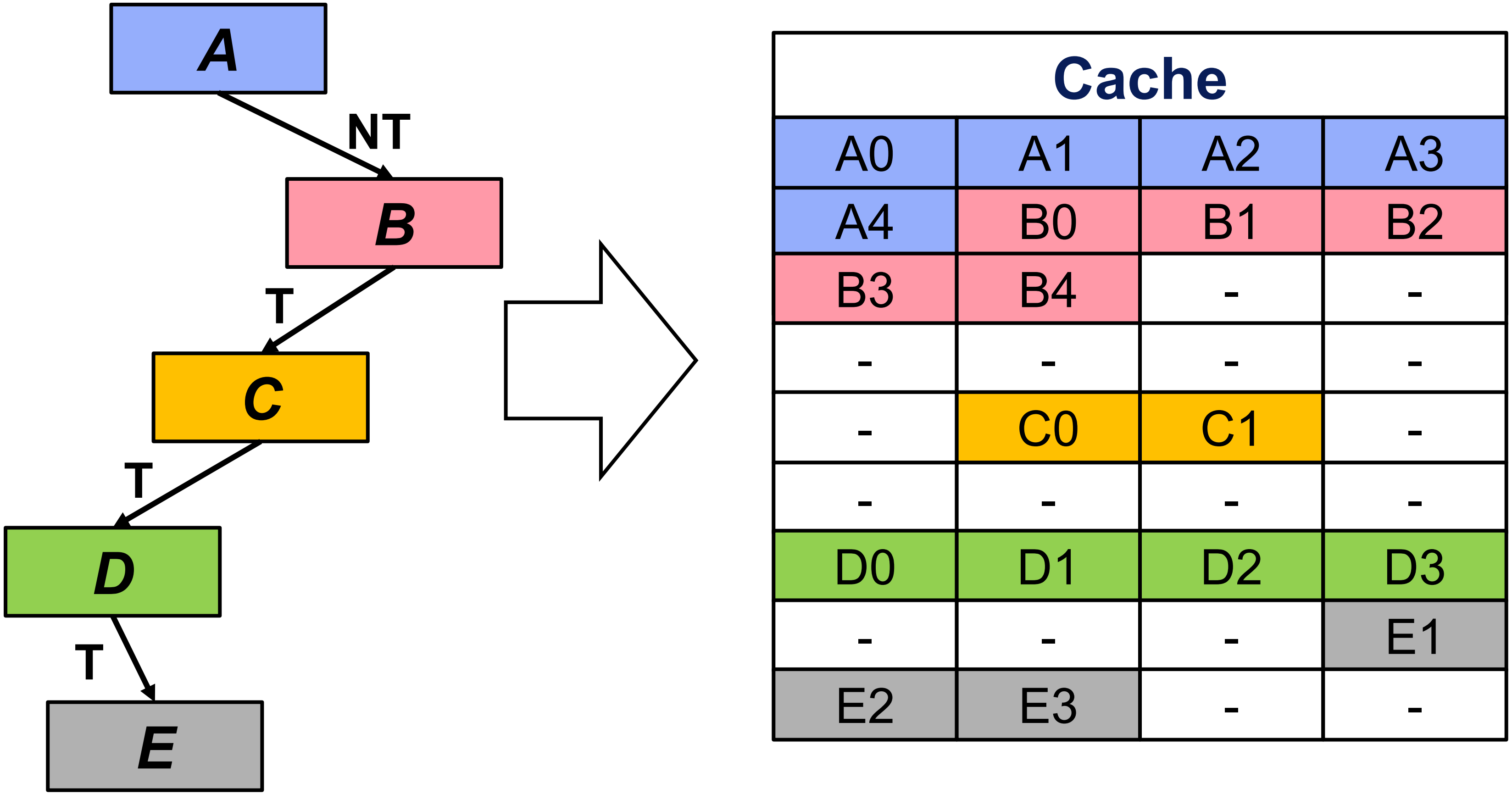


Benchmark	taken %	avg basic block size	# instr between taken branches
eqntott	86.2%	4.20	4.87
espresso	63.8%	4.24	6.65
xlisp	64.7%	4.34	6.70
gcc	67.6%	4.65	6.88
sc	70.2%	4.71	6.71
compress	60.9%	5.39	8.85

Every 4~8 instructions are placed in non-contiguous region

Trace caching [MICRO'96]

- Trace cache allocates instructions from different basic blocks (contiguous instructions) in a single cache line



Cache			
A0	A1	A2	A3
A4	B0	B1	B2
B3	B4	-	-
-	-	-	-
-	C0	C1	-
-	-	-	-
D0	D1	D2	D3
-	-	-	E1
E2	E3	-	-

Trace caching [MICRO'96]

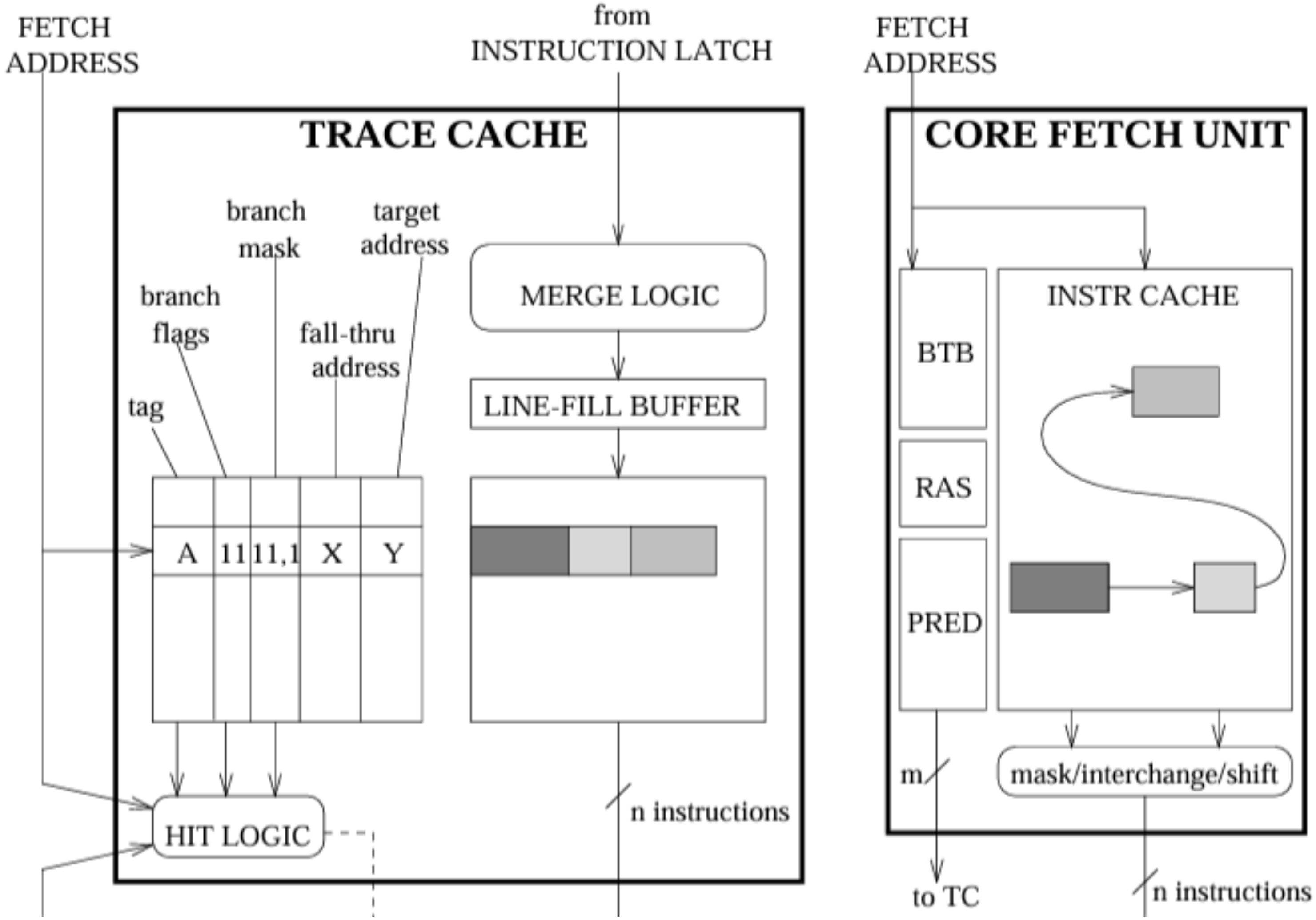
- Trace cache allocates instructions from different basic blocks (contiguous instructions) in a single cache line

Cache			
A0	A1	A2	A3
A4	B0	B1	B2
B3	B4	-	-
-	-	-	-
-	C0	C1	-
-	-	-	-
D0	D1	D2	D3
-	-	-	E1
E2	E3	-	-

Trace Cache			
Metadata: A0, 0111, …			
A0	A1	A2	A3
A4	B0	B1	B2
B3	B4	C0	C1
D1	D2	D3	D4
E1	E2	E3	-
-	-	-	-
-			-
-			-

Use this data if the predictions are 0 -> 1 -> 1 -> 1

Trace caching [MICRO’96]



Question?

Announcements:

Reading: *finish reading P&H Ch.4*

Handouts: *none*