Communications and Computer Engineering II:

Microprocessor 2: Processor Micro-Architecture

Lecturer : Tsuyoshi Isshiki

Dept. Information and Communications Engineering,

Tokyo Institute of Technology

isshiki@ict.e.titech.ac.jp

Lecture Outline

1. Looking back at CISC vs. RISC
2. Processor micro-architecture
3. CISC microcode architecture
4. RISC Processor pipeline
5. Cache memory
6. Instruction-level parallelism

Looking Back at CISC vs. RISC

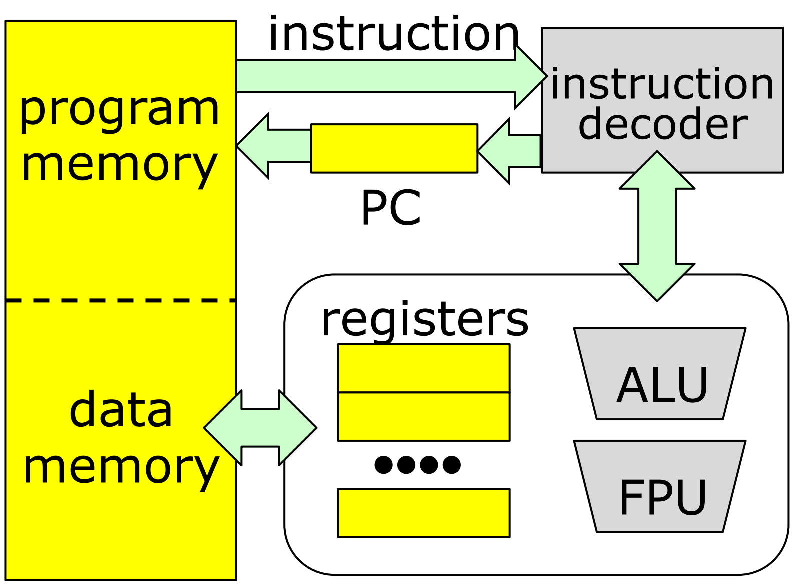
1) **CISC:** Complex Instruction-Set Computer (Intel x86)
   - ***Variable*** instruction length → complex instruction decoder
   - Rich addressing mode : many ways to access memory
2) **RISC:** Reduced Instruction-Set Computer (MIPS, ARM)
   - ***Fixed*** instruction length → simple instruction decoder
   - ***Memory access*** : load or store only
   - Compute operands & results : registers

- **Early computers were all CISC : WHY?**
  - Programs were most written in assembly language → good compilers were not available at that time
  - Made sense to put more functionality into each instruction → variable instruction length, rich addressing mode
- **What "were" the essential CISC building blocks?**

  → **Microcode-based controller**

Processor Micro-Architecture

- **Micro-architecture:** detail hardware architecture and behavior of the instruction execution flow
- All CISC architectures used microcode-based controller, so "micro-architecture" used to mean ***microcode* architecture** → today the terminology applies to all kinds of processor architecture (since microcode architectures are *mostly non-existence*)



Control signals to HW:

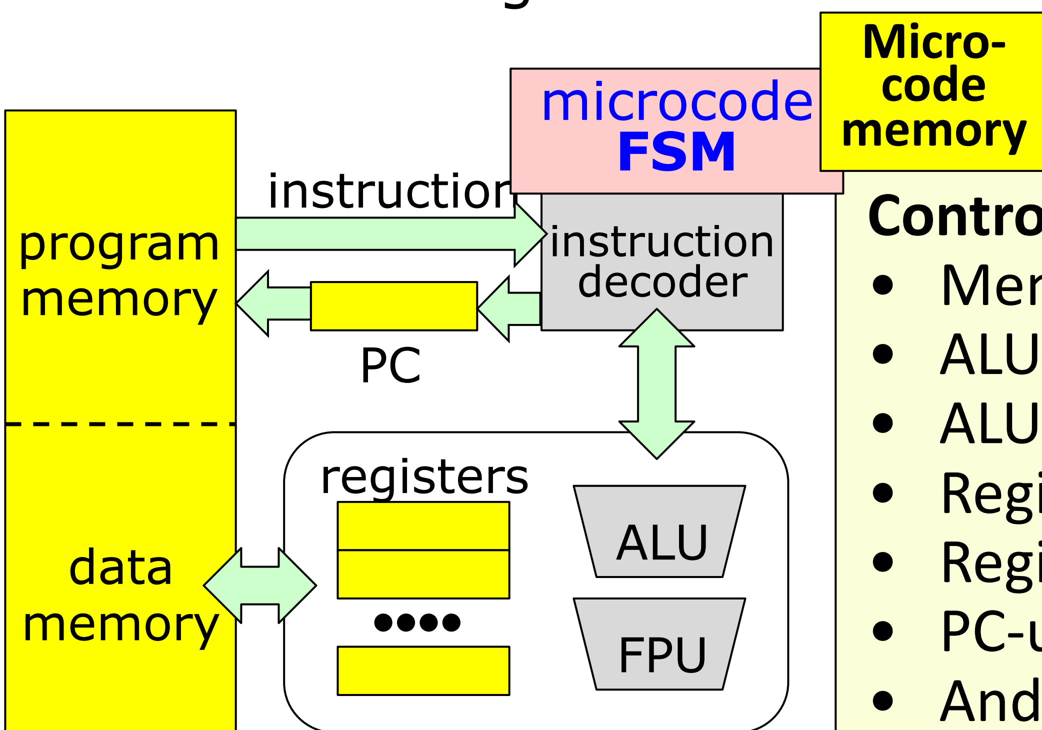
- Memory: R/W, addr, din
- ALU/FPU inputs
- ALU/FPU op-type
- Register inputs, outputs
- Register write enable
- PC-update enable
- And a lot more…

Lecture Outline

1. Looking back at CISC vs. RISC
2. Processor micro-architecture
3. CISC microcode architecture
4. RISC Processor pipeline
5. Cache memory
6. Instruction-level parallelism

CISC Microcode Architecture

- **Microcode:** hardware-level “instructions” for implementing finite state machine(FSM) (“programmable state machine”) that generates all control signals to the hardware



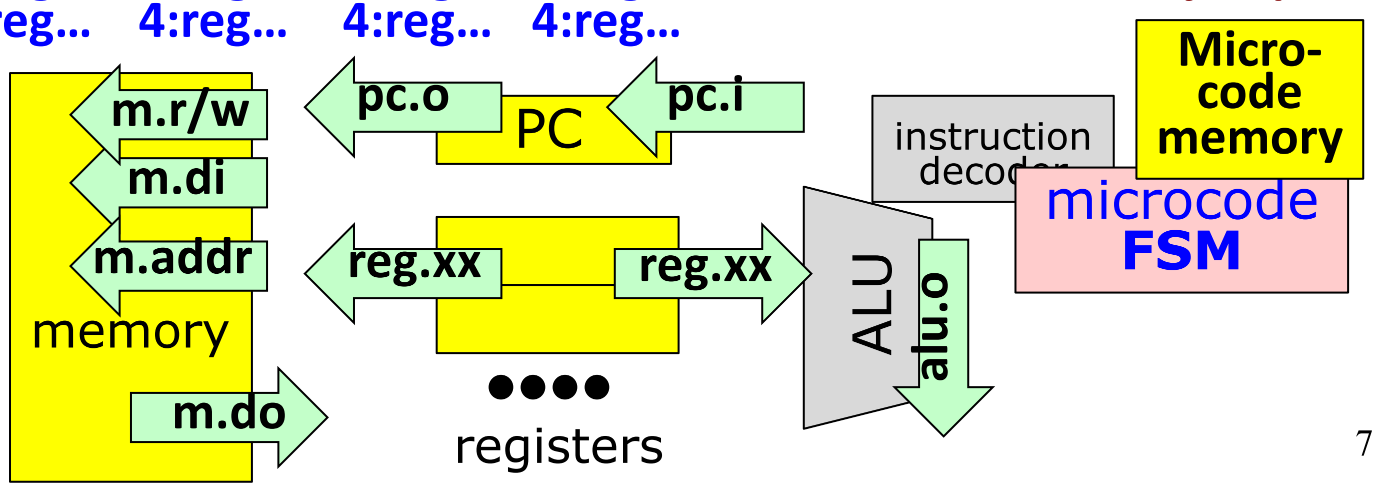
Control signals to HW:

- Memory: R/W, addr, din
- ALU/FPU inputs
- ALU/FPU op-type
- Register inputs, outputs
- Register write enable
- PC-update enable
- And a lot more…

CISC Microcode Architecture

- **Horizontal microcode:** entries in the microcode represent control signal outputs
- **FSM state transition:** “jmp-type” “jmp-addr” contains the next microcode address and necessary action (next, jump, call, dec-jump)

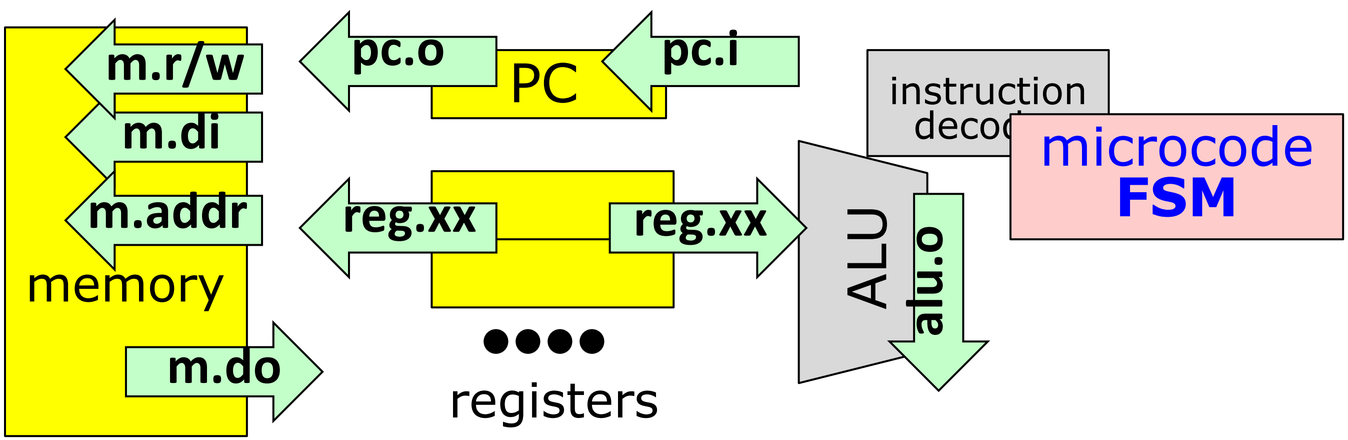
m.r/w	m.addr	m.di	alu.i0	alu.i1	alu.op	●●	jmp-type	jmp-addr
0: none	0:pc.o	0:pc.o	0:m.do	0:m.do	0:NOP		0: next	
1: read	1:alu.o	1:alu.o	1:reg.0	1:ir.imm	1:ADD		1: jump	
2: write	2:reg.0	2:reg.0	2:reg.1	2:reg.0	2: …..		2: call	
	3:reg.1	3:reg.1	3:reg.2	3:reg.1			3: dec-jump	
	4:reg…	4:reg…	4:reg…	4:reg…				



Instruction Execution Cycle

1. **Fetch:** read instruction at PC
2. **Decode:** decode instruction
   a) PC ← PC + 1
3. **Execute** (ADD: M[reg] ← M[reg] + imm)
   a) Load: alu.i0 ← M[ir.ra]
   b) Move: alu.i1 ← ir.imm
   c) Add: alu.o ← alu.i0 + alu.i1
   d) Store: M[ir.ra] ← alu.o

ir: instruction register
reg-ID → ir.ra
imm → ir.imm



Microprogram

m.r/w	m.addr	m.di	alu.i0	alu.i1	alu.op	••	jmp-type	jmp-addr

FETCH: fetch and decode-jump

read	pc.o	*	*	*	*	••	next	*
*	*	*	*	*	*	••	dec-jump	JP_TBL

Load instruction

PC ← PC + 1 (not shown here)

ADD: M[reg] ← M[reg] + ir.imm

read	ir.ra	*	*	*	*	••	next	*
none	*	*	m.do	ir.imm	ADD	••	next	*
write	ir.ra	alu.o	*	*	*	••	jump	FETCH

Read mem-operand

Store ALU output

Set ALU input ports & op

a) Load: alu.i0 ← M[ir.ra]
b) Move: alu.i1 ← ir.imm
c) Add: alu.o ← alu.i0 + alu.i1
d) Store: M[ir.ra] ← alu.o

ADD finished…
Go back to FETCH

Instruction Decode and Jump

m.r/w	m.addr	m.di	alu.i0	alu.i1	alu.op	●●	jmp-type	jmp-addr

ir ← M[PC] (not shown here)

FETCH: fetch and decode-jump

read	pc.o	*	*	*	*	●●	next	*
*	*	*	*	*	*	●●	dec-jump	JP_TBL

NOP: JP_TBL + 0

4 microcodes

ADD: JP_TBL + 4

4 microcodes

SUB: JP_TBL + 8

4 microcodes

AND: JP_TBL + 12

●●

If cannot fit within 4 microcodes, use "call"

ir.opcode:

000 → JP_TBL + 0 : NOP
001 → JP_TBL + 4 : ADD
010 → JP_TBL + 8 : SUB
011 → JP_TBL + 12 : AND
. . .

CISC → RISC Transition (Late 1980's)

- **Microcode-based controller:** enabled flexible architecture extension while maintaining instruction-set backward compatibility
  - To gain performance, instructions became even more complex
- **Game-changing technology trends:**
  - **Good compilers began to emerge:** easier to program at high-level language → compilers tend to use only a fraction of the rich instruction-set
  - **VLSI technology:** single-chip "microprocessor", advance in VLSI design CAD tools, faster memory (near-chip cache) → *gain performance through higher clock frequency operation*

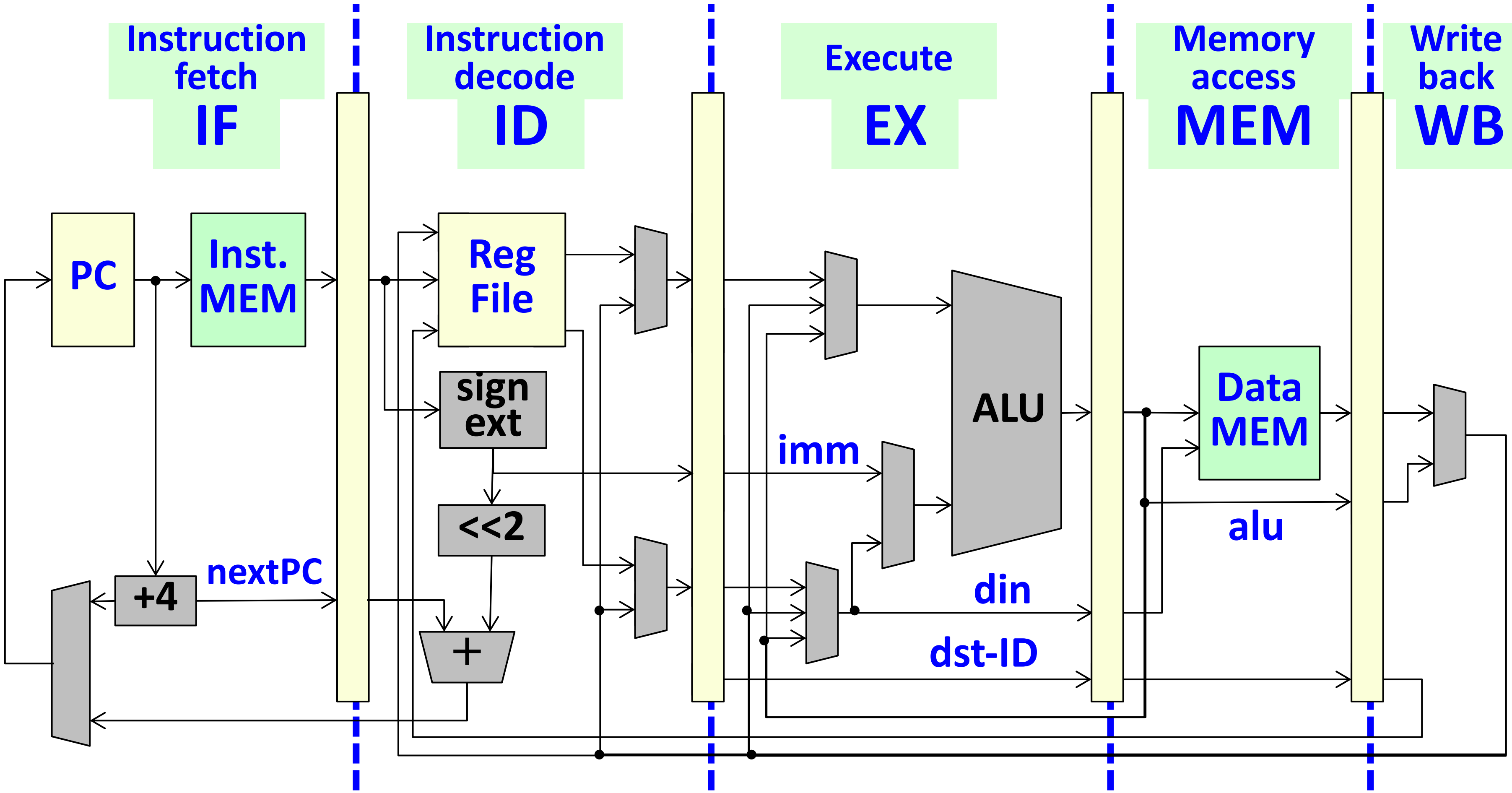
→ **RISC (Reduced Instruction-Set Computer)!!!**

→ **Key architecture feature : PIPELINING!**

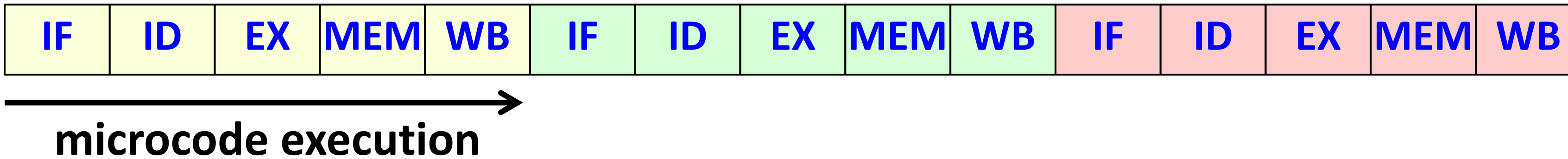
Lecture Outline

1. Looking back at CISC vs. RISC
2. Processor micro-architecture
3. CISC microcode architecture
4. RISC Processor pipeline
5. Cache memory
6. Instruction-level parallelism

MIPS Pipeline

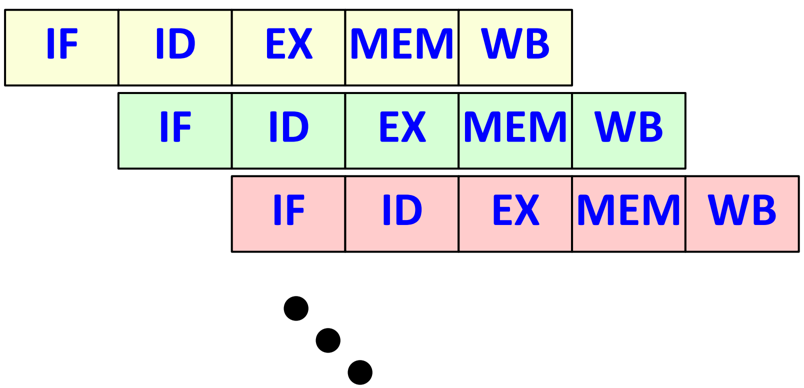


CISC Microcode vs. RISC Pipeline



CISC Microcode-based control

- Requires multiple cycles per instruction



RISC pipeline

- **1 instruction per cycle** (ideal case) → *non-ideal cases :* ***pipeline hazards***
- **Hard-wired FSM** control → simple instructions

Pipeline Hazards

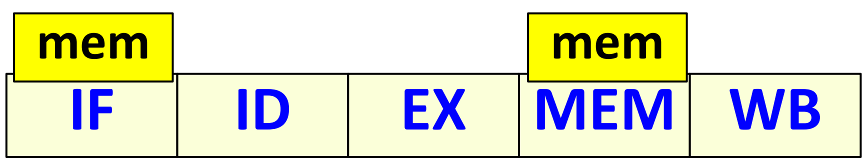
- **Data hazards:** read-after-write (RAW) dependency on a register among multiple instructions

```
R1 ← R1 + R2
R3 ← R1 + R3
```

- **Control hazards:** dependency on PC in conditional branch

```
if (R1 == 0) PC ← PC + imm16
```

- **Structural hazards:** HW resource shared by multiple stages

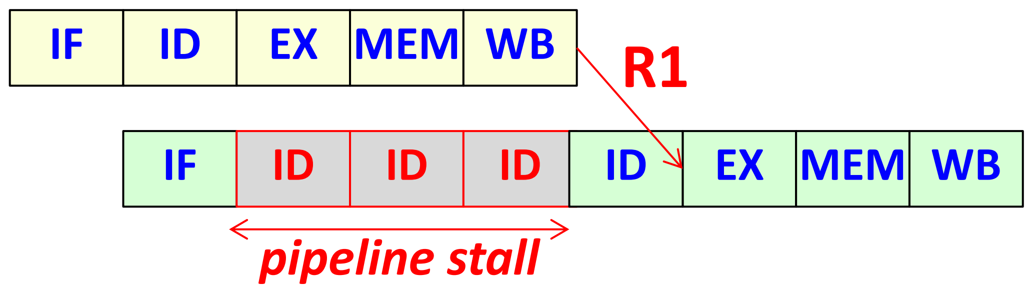


Data Hazards

- **Data hazards:** read-after-write (RAW) dependency on a register among multiple instructions → ***how to avoid pipeline stall ?***

R1 ← R1 + R2
R3 ← R1 + R3

3 stall cycles



R1 ← R1 + R2
R2 ← R2 + R3
R3 ← R1 + R3

2 stall cycles

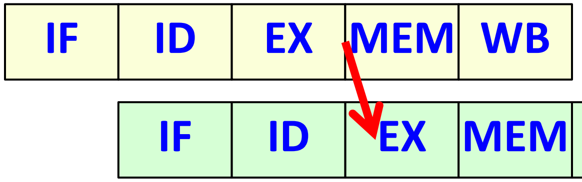
R1 ← R1 + R2
R2 ← R2 + R3
R4 ← R3 + R5
R3 ← R1 + R3

1 stall cycles

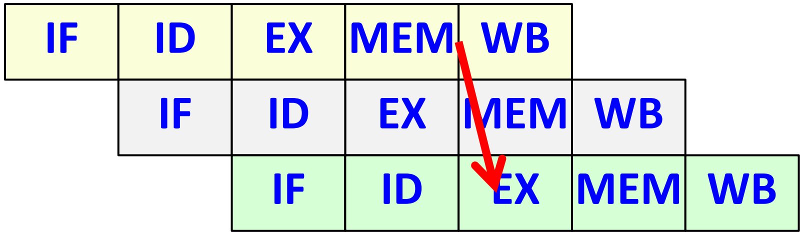
Data Forwarding

- Directly forward the data from MEM/WB stages to DC/EX stage

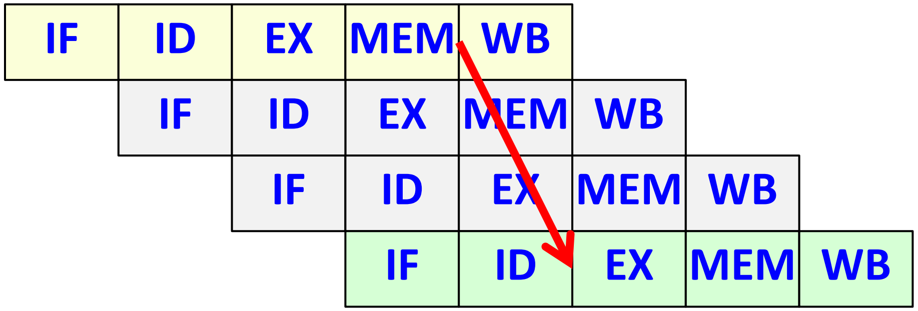
R1 ← R1 + R2
R3 ← **R1** + R3



R1 ← R1 + R2
R2 ← R2 + R3
R3 ← **R1** + R3



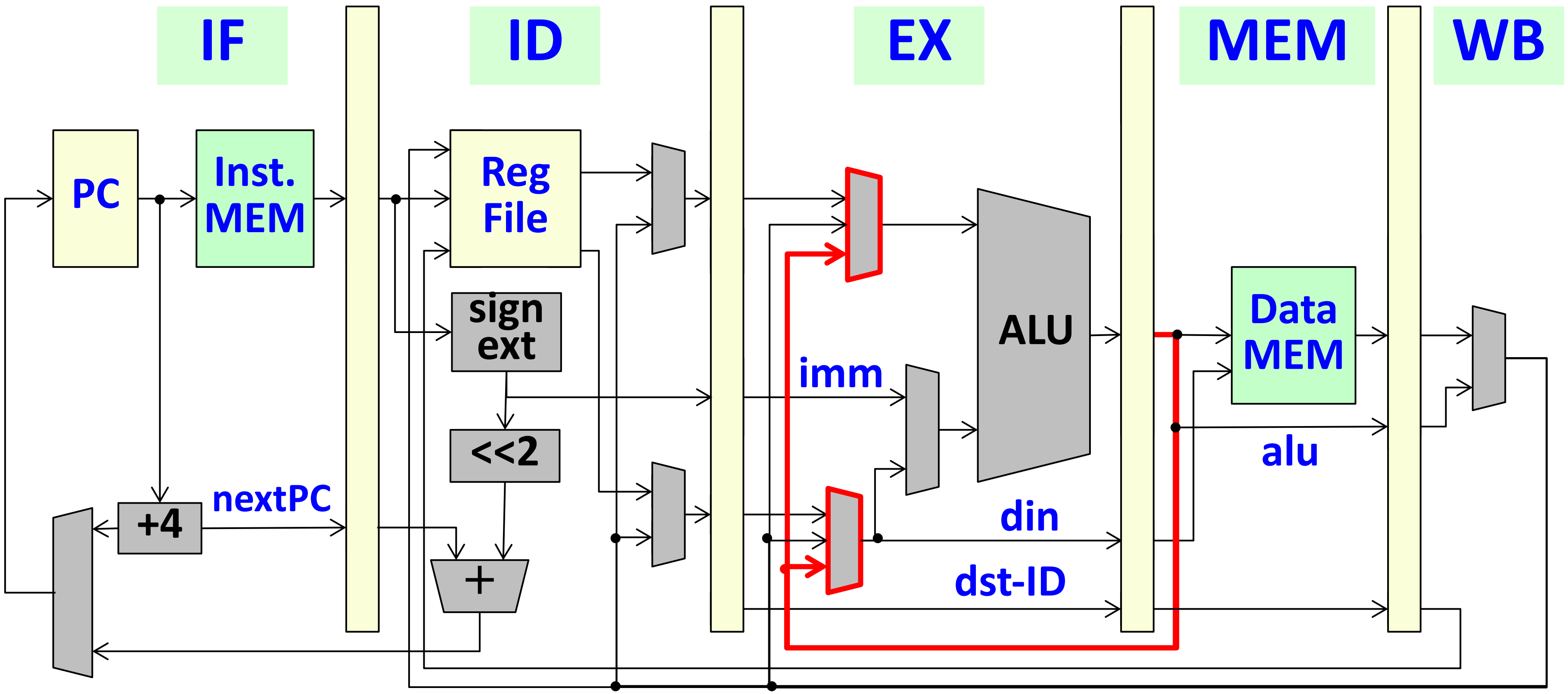
R1 ← R1 + R2
R2 ← R2 + R3
R4 ← R3 + R5
R3 ← **R1** + R3



Data Forwarding Paths

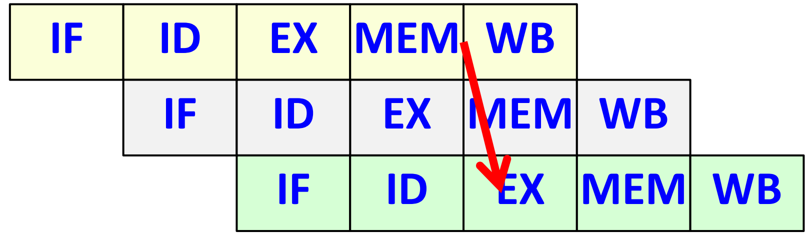
R1 ← R1 + R2
R3 ← R1 + R3

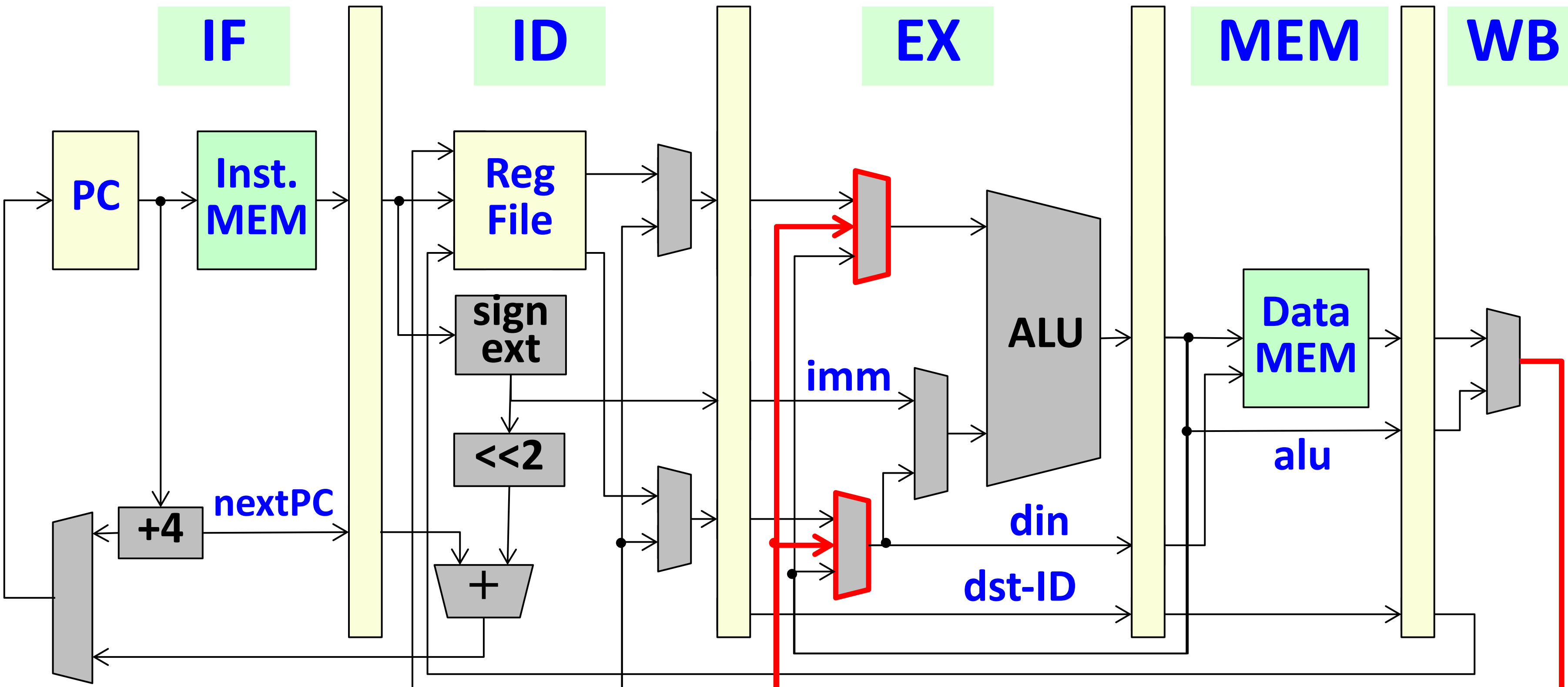
IF	ID	EX	MEM	WB	
	IF	ID	EX	MEM	WB



Data Forwarding Paths

R1 ← R1 + R2
R2 ← R2 + R3
R3 ← **R1** + R3

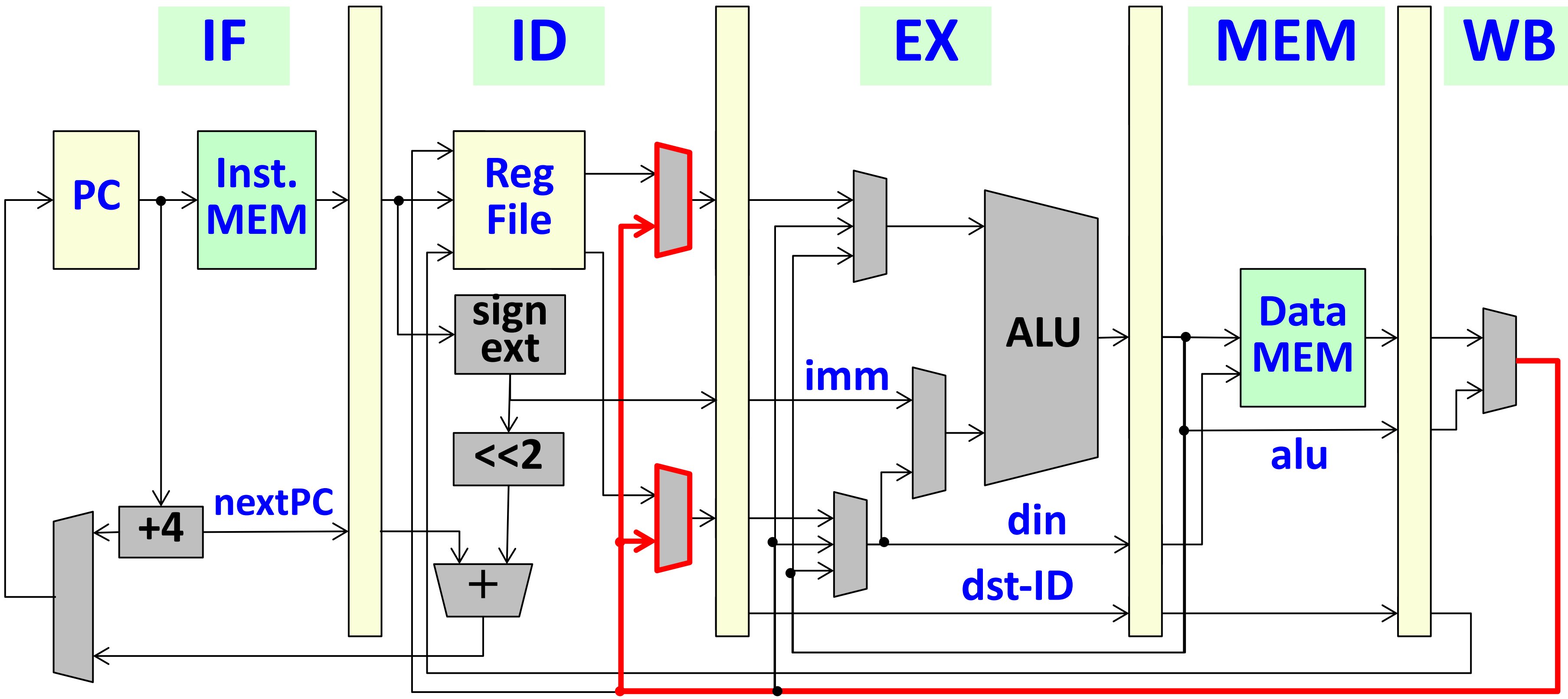




Data Forwarding Paths

R1 ← R1 + R2
R2 ← R2 + R3
R4 ← R3 + R5
R3 ← **R1** + R3

IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	WB



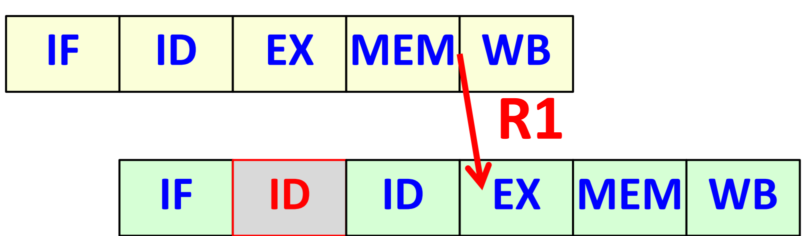
Load Hazards

- **Load hazards:** read-after-write (RAW) dependency after LOAD instruction

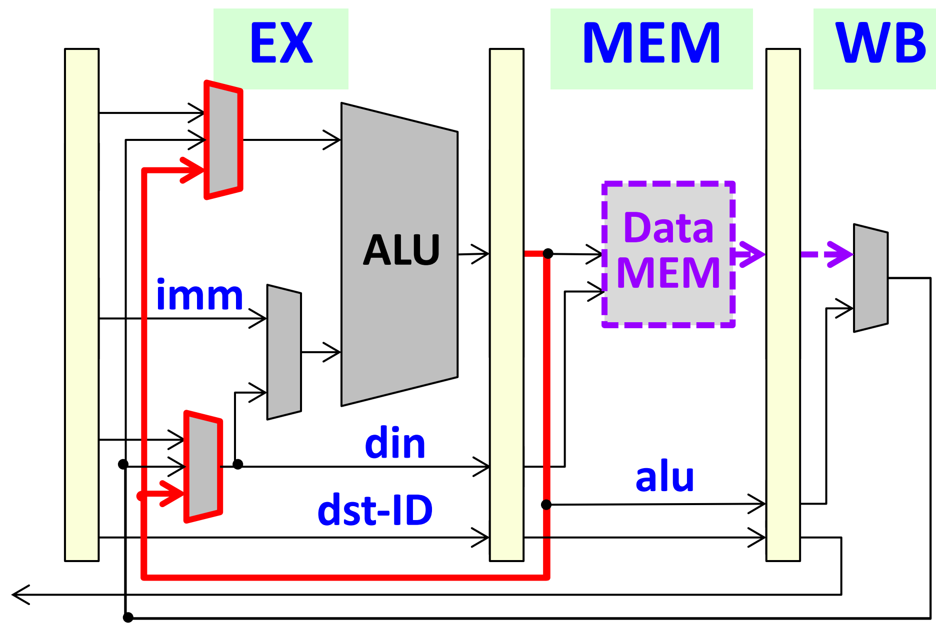
→ Load data becomes available only after MEM stage

R1 ← M[R2]
R3 ← R1 + R3

1 stall cycles



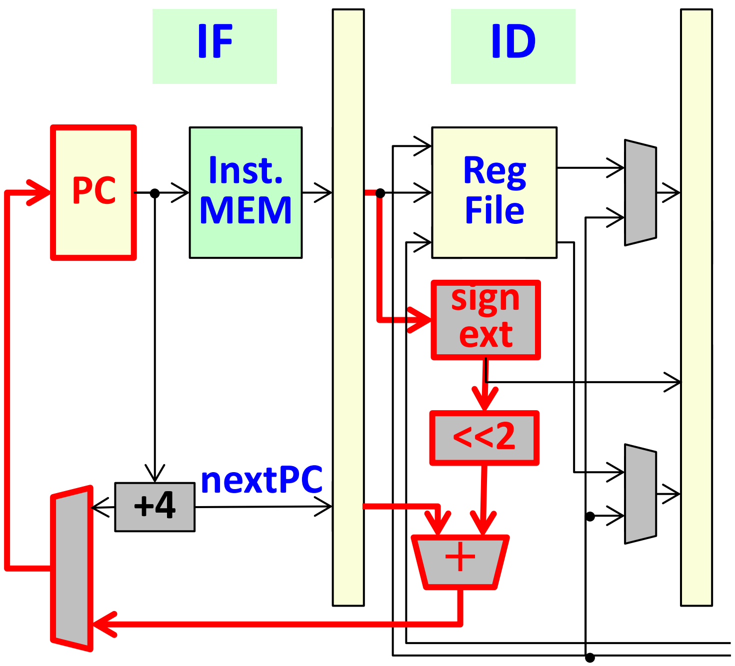
pipeline stall cannot be avoided even with data forwarding

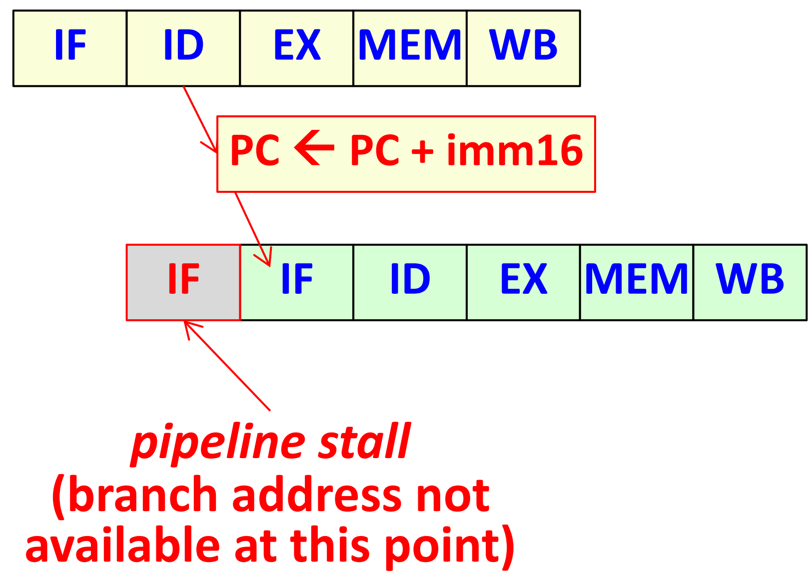


Control Hazards

- **Control hazards:** dependency on PC in conditional branch → ***how to avoid pipeline stall ??***

if (R1 == 0) PC ← PC + imm16





Branch Delay Slot

- Branch delay slot: one or more instructions after the branch instruction (1 delay slot in MIPS case)
  - Instruction at the delay slot is executed regardless of the branch outcome
  - Safest way is to put a **NOP** in the delay slot
  - Clever use of delay slot decreases branch penalty

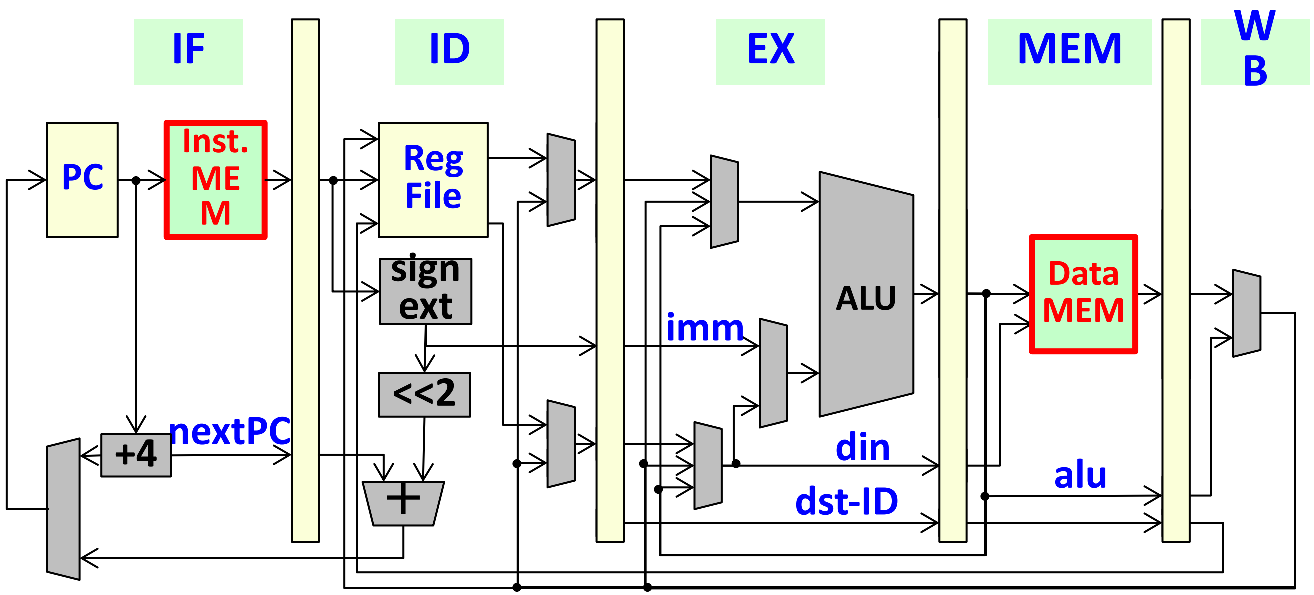
R2 ← R1 + R3;
if (R1 == 0) goto L1;
NOP; // delay slot

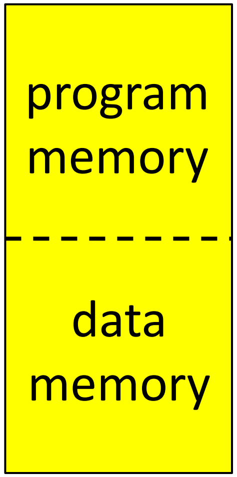
if (R1 == 0) goto L1;
R2 ← R1 + R3;

- Branch delay slots can be filled by instruction in the *branch-taken path* or *branch-not-taken path*, as long as these instructions does not have side effects on the other branch path

Structural Hazards

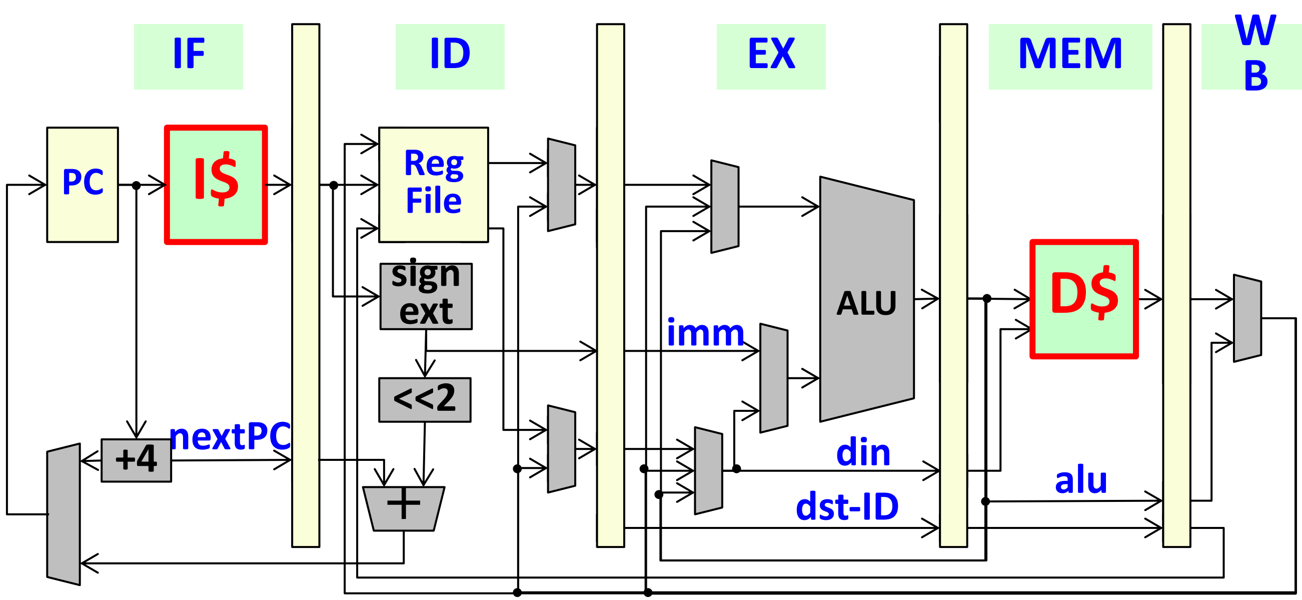
- HW resource shared by multiple stages
  - **Multi-cycle instruction** (such as DIV): occupies EX stage continuously, preventing the following instruction to enter EX stage
  - **Memory:** simultaneous access can happen at IF and MEM stages → Can we separate instruction memory and data memory???

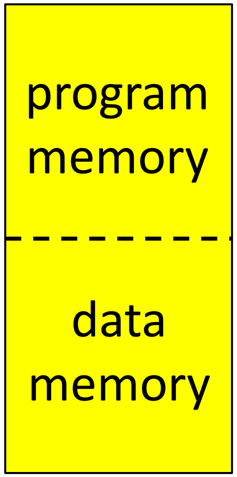




Memory Architecture

- Unified memory architecture: Program memory and data memory located inside the same address (common for general-purpose systems with complex SW layers)
- Harvard architecture: separate program memory and data memory → common for DSPs
- **Cache memory** allows physical separation on unified memory architecture





Lecture Outline

1. Looking back at CISC vs. RISC
2. Processor micro-architecture
3. CISC microcode architecture
4. RISC Processor pipeline
5. Cache memory
6. Instruction-level parallelism

Cache Memory

- Cache memory : fast/small SRAM (near-chip or on-chip) → smaller memory is faster in general
  - Cache hit : accessed memory word exists in cache → cache is effective only if cache hit-rate is high
- Relies on two "locality" properties
  - **Temporal locality :** accessed memory word will likely to be accessed again very soon → make sense to put the accessed memory word to a faster cache
  - **Spatial locality :** accessed memory word will likely have its "neighboring words" accessed again very soon → make sense to put *multiple words* on the cache at once

Cache Memory

- Cache line:
  - Data : 32 bytes ~ 64 bytes per line (multiple words)
  - Tag : indicates corresponding address of this line
- Associativity (N-way) : any word can be at N different locations in the cache memory
  - N = 1 : direct mapping (any word has a unique location)
- Address partition

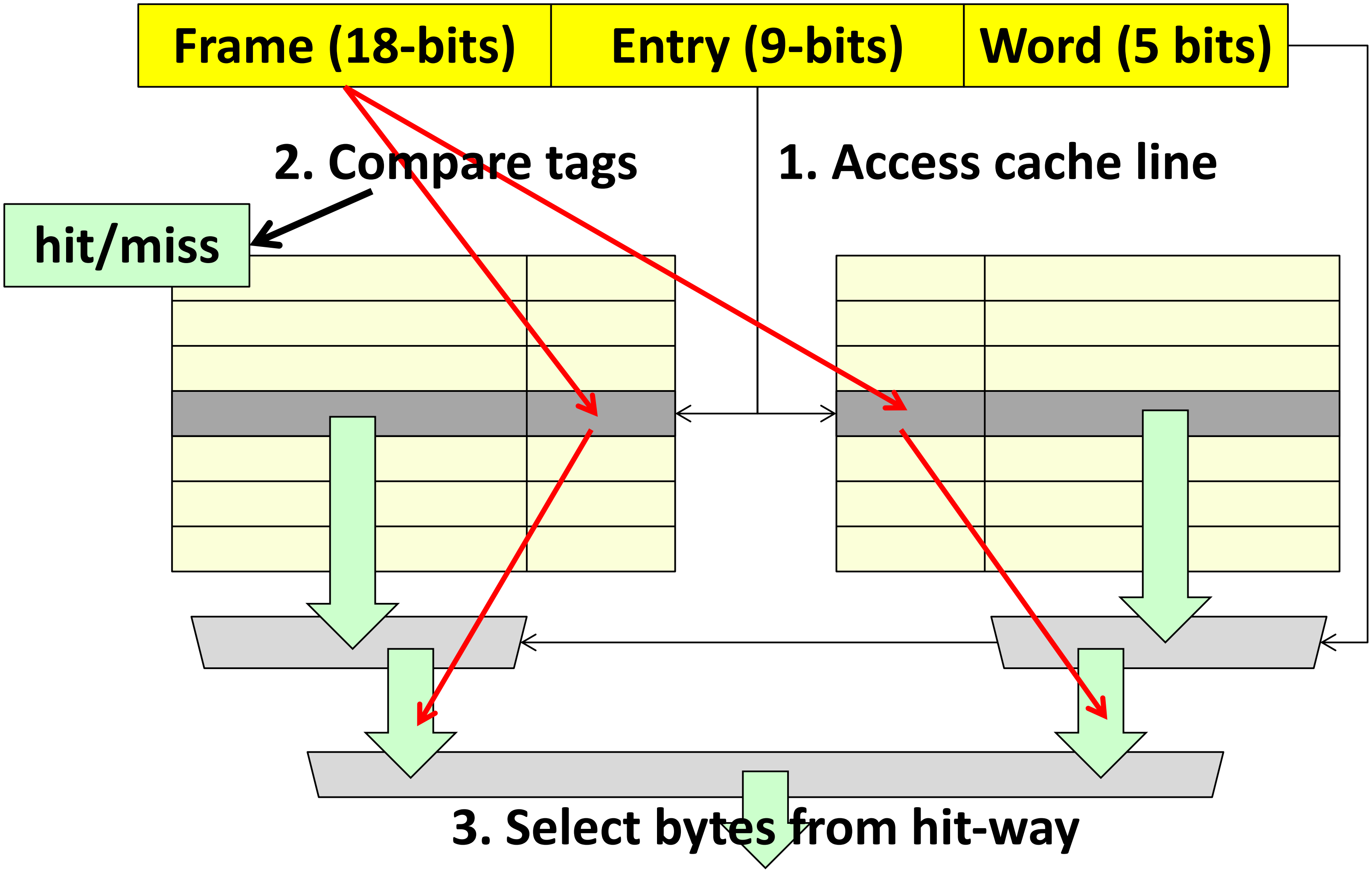
Frame addr	Entry addr	Word addr
Stored in tag	Cache line selection	Word/byte selection

Ex: 32-bit address, 32 bytes/line, 2-way, 32KB size

- **Word addr : 5 bits**
- **Entry addr : 9 bits**
- **Frame addr : 18 bits**

16KB/way = 2^{14} bytes/way
→ entry-addr + word-addr = 14 bits
→ frame-addr = 32 – 14 = 18 bits

Cache Access Flow

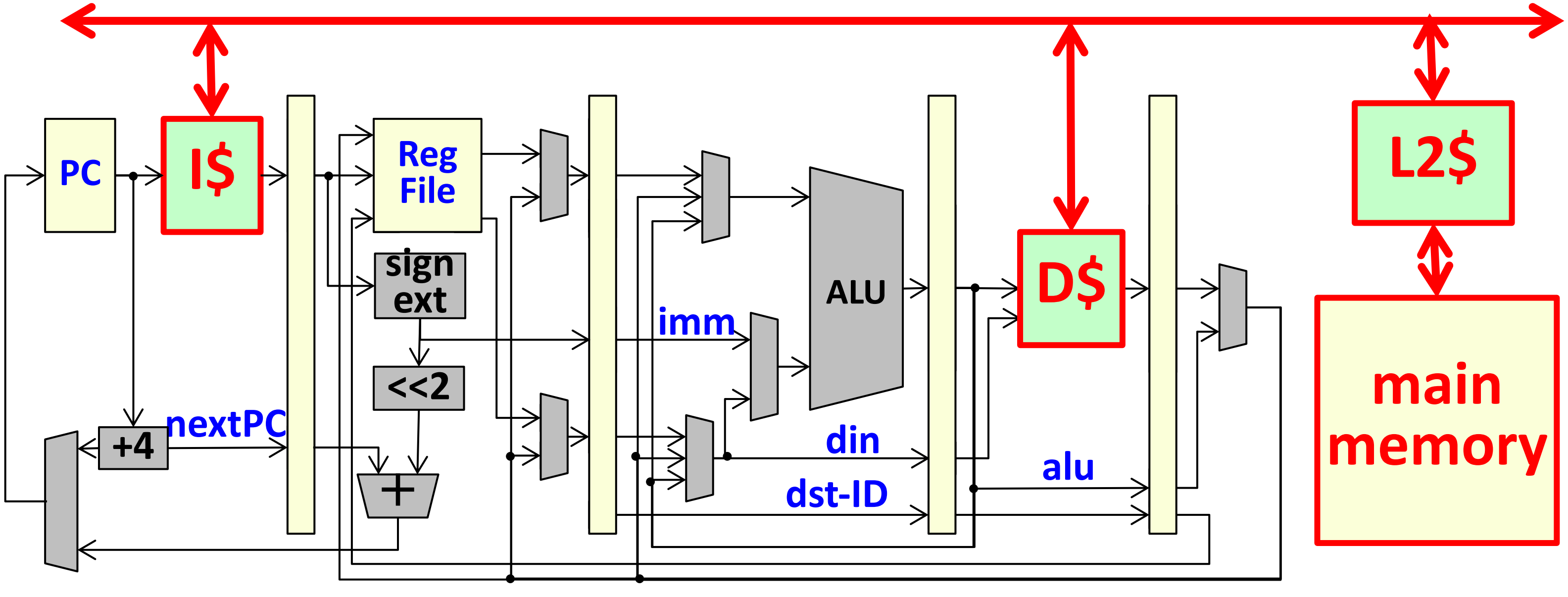


Cache Policies

- **Replacement policy:** which cache line to flush?
  - Random : pick a "way" randomly and replace the entry
  - LRU (least recently used) : replace the entry with the oldest access → complicated implementation if # ways is large
  - Round Robin : rotate replaced ways
- **Write policy:** what to do on cache writes?
  - Write through : always write the new data to main memory → easier to manage "coherency" but bus traffic becomes heavy
  - Write back : write to main memory only when replaced (flushed) → less bus traffic, difficult to maintain "coherency"
- **Coherency:** important for multi-core systems
  - Cache snooping: monitor other cache states individually
  - Directory-based: manage all cache states at one place

Memory Hierarchy

- **Level-1 caches** : on-chip
  - Instruction cache: read-only
  - Data cache: read-write
- **Level-2 caches** : on-chip or off-chip
  - Instruction/data unified cache

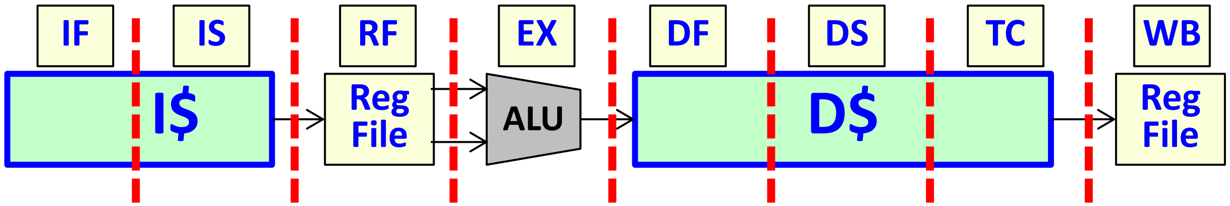


Lecture Outline

1. Looking back at CISC vs. RISC
2. Processor micro-architecture
3. CISC microcode architecture
4. RISC Processor pipeline
5. Cache memory
6. Instruction-level parallelism

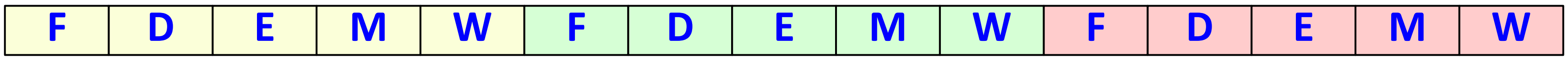
RISC Architecture Enhancement (MIPS case)

- **R2000/R3000** (1985, 1988)**:** 5-stage, I$/D$
- **R4000** (1991): 8-stage ("***super-pipeline***")
  - Additional pipe-stages on I$ and D$
  - Data width : 64-bits

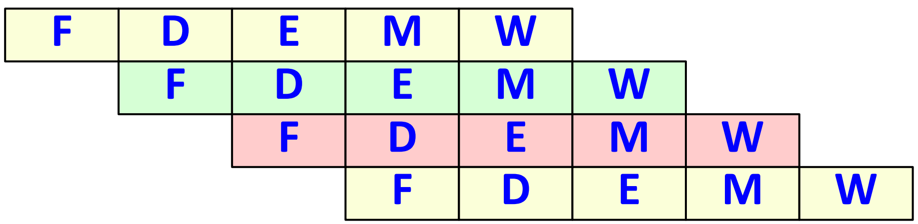


- **R8000**(1994): 4-way *"in-order"* **superscalar**
  - Superscalar: multiple execution pipelines
  - In-order: instructions are issued and completed in order
- **R10000** (1996): 4-way *"out-of-order"* **superscalar**
  - Out-of-order: instructions are issued and completes out of order
  - Techniques: register renaming, instruction reorder buffer, branch prediction (speculative execution), etc.

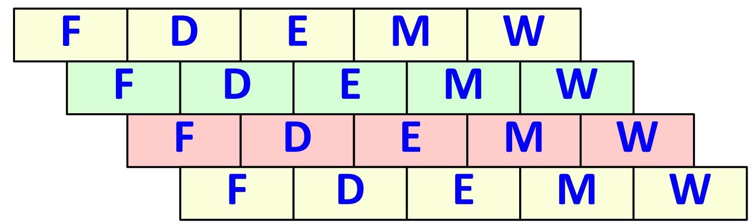
Instruction-Level Parallelism



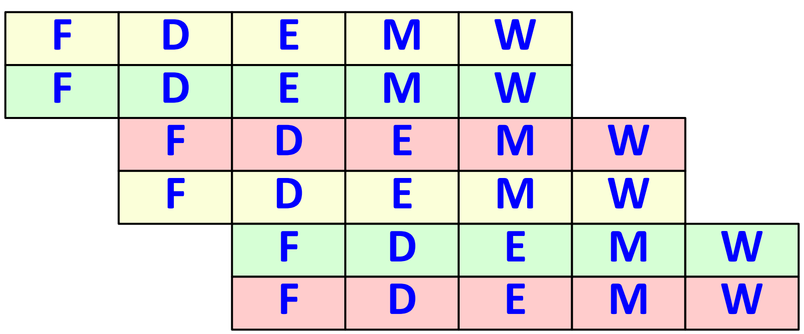
Sequential (microcode-based FSM)



"Scalar" Pipeline



Super-pipeline



Superscalar pipeline

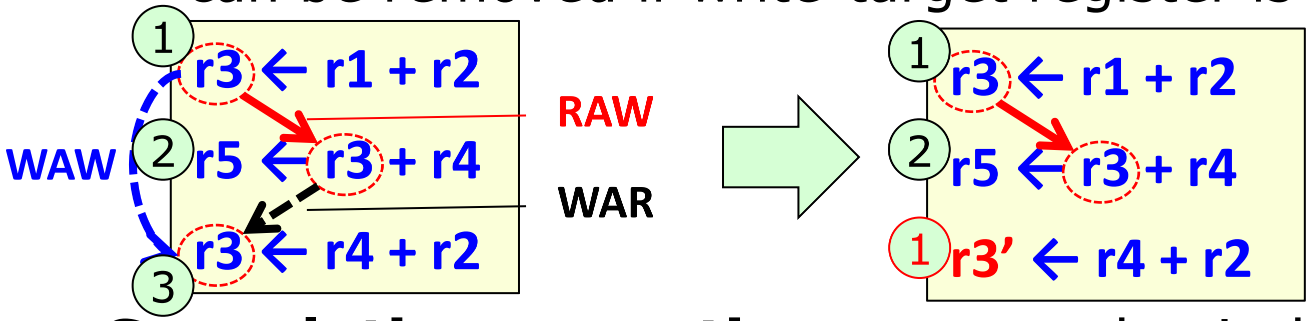
Superscalar: multiple pipelines

- Works well if enough instruction-level parallelism exists
- What limits instruction-level parallelisms?

→ *Data dependencies, branches*

Enhancing Instruction-Level Parallelism

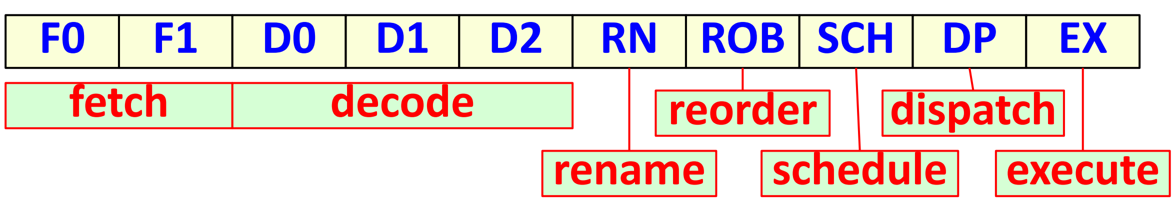
- **Register renaming:** remove "artificial" dependencies
  - Write-after-write(WAW)/Write-after-read(WAR) : can be removed if write-target register is renamed



- **Speculative execution:** remove basic-block boundaries by branch prediction
  - Branch prediction : predict "taken" or "not-taken" based on branch history and other information
  - Speculative execution : continue execution along the predicted branch path, discard the results in case of mis-prediction

Enhancing CISC Architecture (x86 case)

- **8086** (1978)**:** 16-bit machine
- **80186/80286** (1982, 1984): 24-bit address space
- **80386** (1985): 32-bit machine
- **80486** (1989): on-chip cache/FPU, ***5-stage pipeline***
  → microcode-base to hard-wired control
- **Pentium (P5)** (1993): 2-way "*in-order*" **superscalar**
- **Pentium Pro (P6)** (1995): 3-way "*out-of-order*" **superscalar**
  - **Micro-operation :** decompose CISC instruction into "RISC-like" sub-instructions (***hard-wired microcode!!***)
  - Register renaming, speculative execution
  - 10 ~ 14 pipe-stages → ***super-pipeline***



Enhancing CISC Architecture (x86 case)

- **Pentium 4 (NetBurst)** (2000~)**: Hyper-pipeline**
  - 20 pipe-stages (Willamette: 2000)
  - 20 pipe-stages (Northwood: 2002) : **Hyper-threading** (simultaneous multi-threading)
  - 31 pipe-stages (Prescott: 2004) : 64-bit extension
  - Netburst roadmap : 40~50 pipe-stages, 10GHz clock → ***abandoned due to thermal and power issues***
- **Pentium M** (2003): enhancement of **P6 architecture**
  - Better thermal and power efficiency than **NetBurst**
- **"Intel Core"** (2006~present): **multi-cores**
  - **Core Duo :** Dual core, enhancement of **Pentium M** (32-bit)
  - **Core 2 Duo :** Dual core, 64-bit extension
  - **Core i3/i5/i7 : Nehalem micro-architecture** (hyper-threading, dual/quad/octal cores)

X86 Micro-Architectures

1978	8086	16-bits	~10MHz	29,000 Tr.
1982	80286	16-bits	~25MHz	134,000 Tr.
1985	80386	32-bits	~40MHz	275,000 Tr.
1989	80486	32-bits	~150MHz	1.2M Tr.
1993	Pentium	32-bits	~233MHz	3.1M ~ 4.5M Tr.
1995	Pentium-Pro	32-bits	~200MHz	5.5M Tr.
1997	Pentium II	32-bits	~450MHz	7.5M Tr.
1999	Pentium III	32-bits	450M~1.4GHz	9.5M ~ 21M Tr.
2000	Pentium 4	32/64-bits	1.3~3.8GHz	42M ~ 184M Tr.
2003	Pentium M	32-bits	900M~2.6GHz	140M Tr.
2006	Core 2	64-bits	1GHz~3.3GHz	169M ~ 411M Tr.
2008	Core i7	64-bits	~3.2GHz	731M Tr.

CISC vs. RISC as of Today

1) **RISC:**
   - (Scalar) pipeline → super-pipeline → superscalar (in-order → out-of-order)
   - Register renaming, instruction reorder buffer, branch prediction, speculative execution
2) **CISC:**
   - Microcode-based FSM → hard-wired control pipelining → superscalar (in-order → out-of-order)
   - CISC instruction → decomposition into RISC-like *micro-operations*

- Difference of CISC vs. RISC on the instruction-set level is no longer apparent on the micro-architecture level → both uses ***out-of-order superscalar super-pipelines***

Summary

1. Processor micro-architecture
2. CISC microcode architecture
3. RISC Processor pipeline
4. Cache memory
5. Instruction-level parallelism
   - Super-pipeline
   - Superscalar : in-order, out-of-order
6. Instruction execution flow
7. Computer arithmetics
8. CISC vs. RISC (today)
   - Difference in instruction-set architectures is not apparent in micro-architectures

Report Submission

- What will be the most important market(s) for microprocessors ***10 years from now*****?**
- Which instruction-set architecture (existing or new) will dominate this market(s)?
- What will be the deciding factor (key features) of that dominating architecture?

→ Describe your original views covering the technical aspects including hardware architecture design issues, software issues (compilers, programming language, platforms), and manufacturing issues.

→ **Deadline : One Week from TODAY !!**

→ Submit your report (MS-WORD or PDF) by email to:

isshiki@ict.e.titech.ac.jp
Subject: ICT-II Report submission