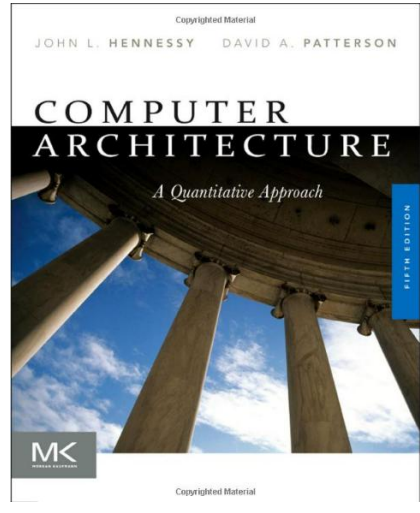
แก้ไขครั้งล่าสุดเมื่อวันที่ 29 ตุลาคม 2568

Chapter 12

Microprocessor without Interlocked Pipeline Stages (MIPS)

Not to be confused with millions of instructions per second



ประเด็น	สแตกซีพียู	แอลซีทรี
เรจิสเตอร์	ไม่มีเรจิสเตอร์สำหรับใช้งานทั่วไป ใช้สแตก	มีเรจิสเตอร์สำหรับใช้งานทั่วไป
ขนาดโปรแกรม	โปรแกรมหรือ binary code มีขนาดเล็ก เพราะแต่ละคำสั่งไม่ต้องระบุเรจิสเตอร์	โดยทั่วๆ ไป binary code จะมีขนาดใหญ่กว่าโปรแกรมเดียวกันที่เขียนด้วยภาษา assembly ของสแตกซีพียู
การเรียกฟังก์ชัน	สะดวก โปรแกรมไม่ต้องทำ stack frame เอง มีฮาร์ดแวร์สำหรับ data stack และ return stack ให้	ยากมาก โปรแกรมเมอร์ต้องสร้างสแตกเฟรมเอง โดยใช้หน่วยความจำหลัก
การเขียนโปรแกรม	เขียนโปรแกรมด้วยภาษา assembly ได้ง่าย ไม่ต้องคอยจำว่าเรจิสเตอร์ตัวไหนเก็บค่าอะไรอยู่	โปรแกรมด้วยภาษา assembly ยาก เหมาะกับการเขียนโปรแกรมด้วยภาษาระดับสูง แล้วใช้คอมไพเลอร์แปล
การใช้หน่วยความจำ	ไม่คุ้มค่า เพราะแยกเป็น MEM, DS, RS บางครั้ง DS ไม่พอ จะไปยืม MEM มาใช้ก็ไม่ได้	ใช้หน่วยความจำคุ้มค่า เพราะใช้หน่วยความจำหลักชิ้นเดียวสำหรับทำทุกอย่าง ทั้งเก็บข้อมูลและทำสแตก

Complex Instruction Set Computer (CISC)
Reduced Instruction Set Computer (RISC)

The CISC Approach

```
ADD addr1, addr2
```

The RISC Approach

```
LOAD reg1, addr1
LOAD reg2, addr2
ADD reg1, reg2
STORE addr1, reg1
```

https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/

CISC

- Emphasis on hardware
- Includes multi-clock complex instructions
- Memory-to-memory: "LOAD" and "STORE" incorporated in instructions
- Small code sizes, high cycles per ~~second~~ instruction
- Transistors used for storing complex instructions

RISC

- Emphasis on software
- Single-clock, reduced instruction only
- Register to register: "LOAD" and "STORE" are independent instructions
- Low cycles per ~~second~~ instruction, large code size
- Spends more transistors on memory registers

ยุคแรกนิยม CISC เพราะเขียน assembly code ง่าย (ยังไม่มีคอมไพเลอร์) ได้ assembly code ที่ไม่ยาวมาก

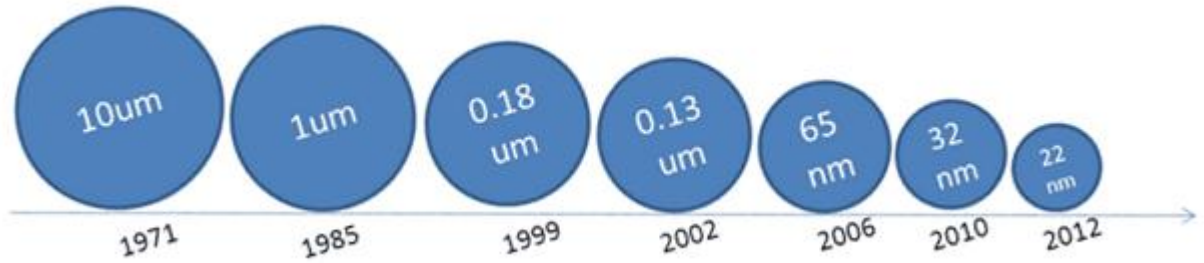
1978 เกิดภาษา C, 1979 Intel ผลิต CPU 8088

Intel 4004, 8008, 8080, 8085, 8086, 8088, 80286, 80386, 80486, Pentium, Atom, Celeron, Xeon Core, Core 2, Core i3, Core i5, Core i7, Core i9

Backward Compatibility หมายถึง CPU รุ่นใหม่ยังรัน executable code เดิมได้
Intel ไม่บอกว่าภายใน processor ออกแบบฮาร์ดแวร์ยังไง บอกแค่ instruction set ว่ามีคำสั่งอะไรบ้าง
x86 หมายถึง สถาปัตยกรรมชุดคำสั่ง 32 บิต x64 หมายถึง สถาปัตยกรรมชุดคำสั่ง 64 บิต

Apple M1

- Designed by Apple Inc., manufactured by TSMC (Taiwan Semiconductor Manufacturing Company Limited)
- Manufacturing technology 5 nm (หรือ process node)
  นิยามอาจจะแตกต่างกันไปในแต่ละโรงงานผลิต เช่น ขนาดของ transistor หรือ ระยะทางระหว่างเส้นลวด 2 เส้น



- Intel ยังอยู่ที่ 10 nm โดน TSMC แซงไปแล้ว, Intel ผลิตเอง แต่บริษัทอื่น ๆ เช่น Apple, Samsung จ้าง TSMC ผลิต
- Apple has a perpetual license on the ARM instruction set, but Apple pays a royalty per chip.

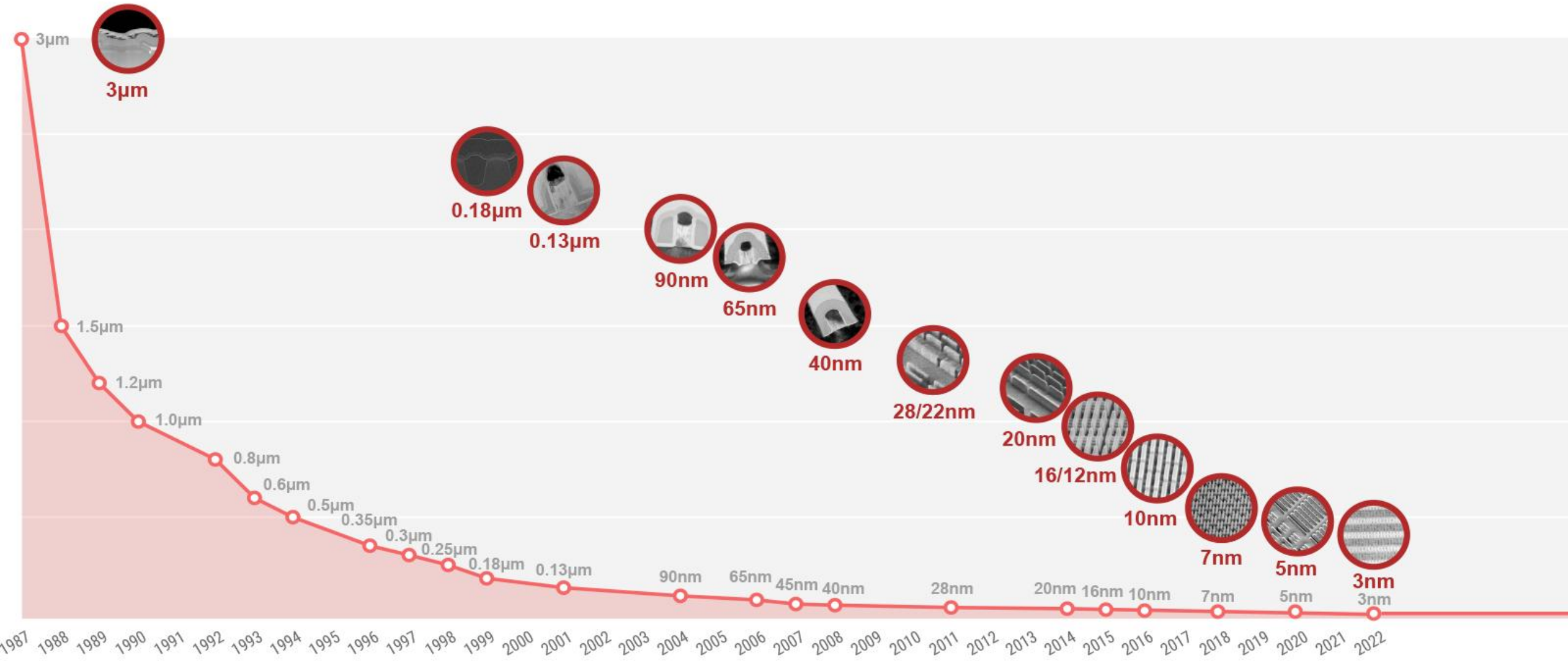
RISC Roadblocks

- A lack of software support. Windows OS does not support RISC
- Although improving the performance of CISC is more difficult, Intel has plenty of resources

RISC Supporting Factors

- The price of RAM has dramatically increased (ไฟล์ exe มีขนาดใหญ่ได้แล้ว)
- Compiler technology has become more sophisticated (แปลภาษาระดับสูงเป็นคำสั่งของ RISC ได้)

TSMC


3μm
1.5μm
1.2μm
1.0μm
0.8μm
0.6μm
0.5μm
0.35μm
0.3μm
0.25μm
0.18μm
0.13μm
90nm
65nm
45nm
40nm
28nm
20nm
16nm
10nm
7nm
5nm
3nm
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022

FEATURE | SEMICONDUCTORS

THE NANOSHEET TRANSISTOR IS THE NEXT (AND MAYBE LAST) STEP IN MOORE'S LAW

Nanosheet devices are scheduled for the 3-nanometer node as soon as 2021

A16 Technology

TSMC A16™ technology is the next nanosheet-based technology featuring Super Power Rail, or SPR. SPR is an innovative, best-in-class backside power delivery solution. It improves logic density and performance...

2nm Technology

TSMC 2nm (N2) technology development is on track and made good progress. N2 technology features the company's first generation of nanosheet transistor technology with full-node strides in performance...

angstrom / angström / ångström / Å = 1.0×10^{-10} meters

https://spectrum.ieee.org/the-nanosheet-transistor-is-the-next-and-maybe-last-step-in-moores-law

MIPS (Microprocessor without Interlocked Pipelined Stages)

- A family of reduced instruction set computer (RISC)
- Computer architecture courses in universities often study the MIPS architecture
- MIPS architecture was pioneered at Stanford University
- MIPS Computer Systems, Inc. was founded in 1984

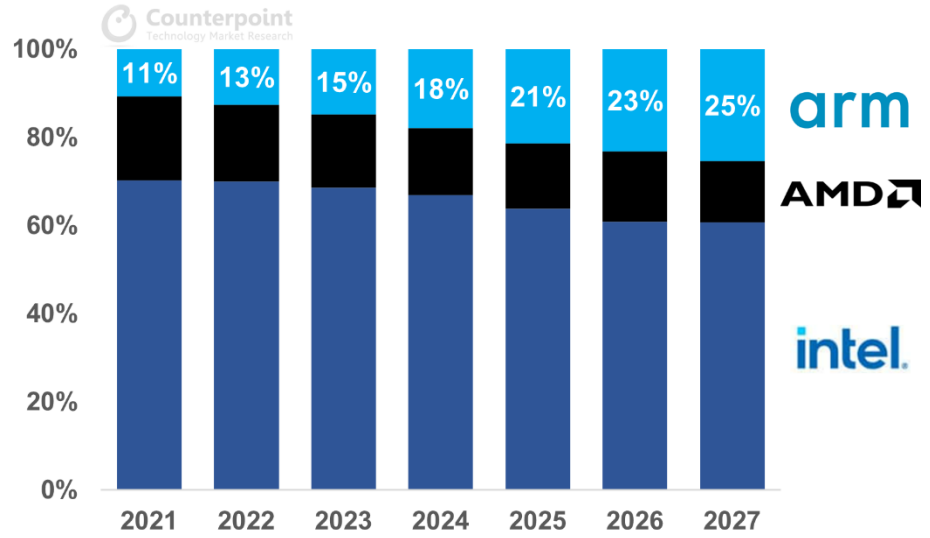
RISC-V (pronounced risk-five)

- Open standard instruction set architecture
- Royalty-free open-source licenses
- MIPS architecture had ended

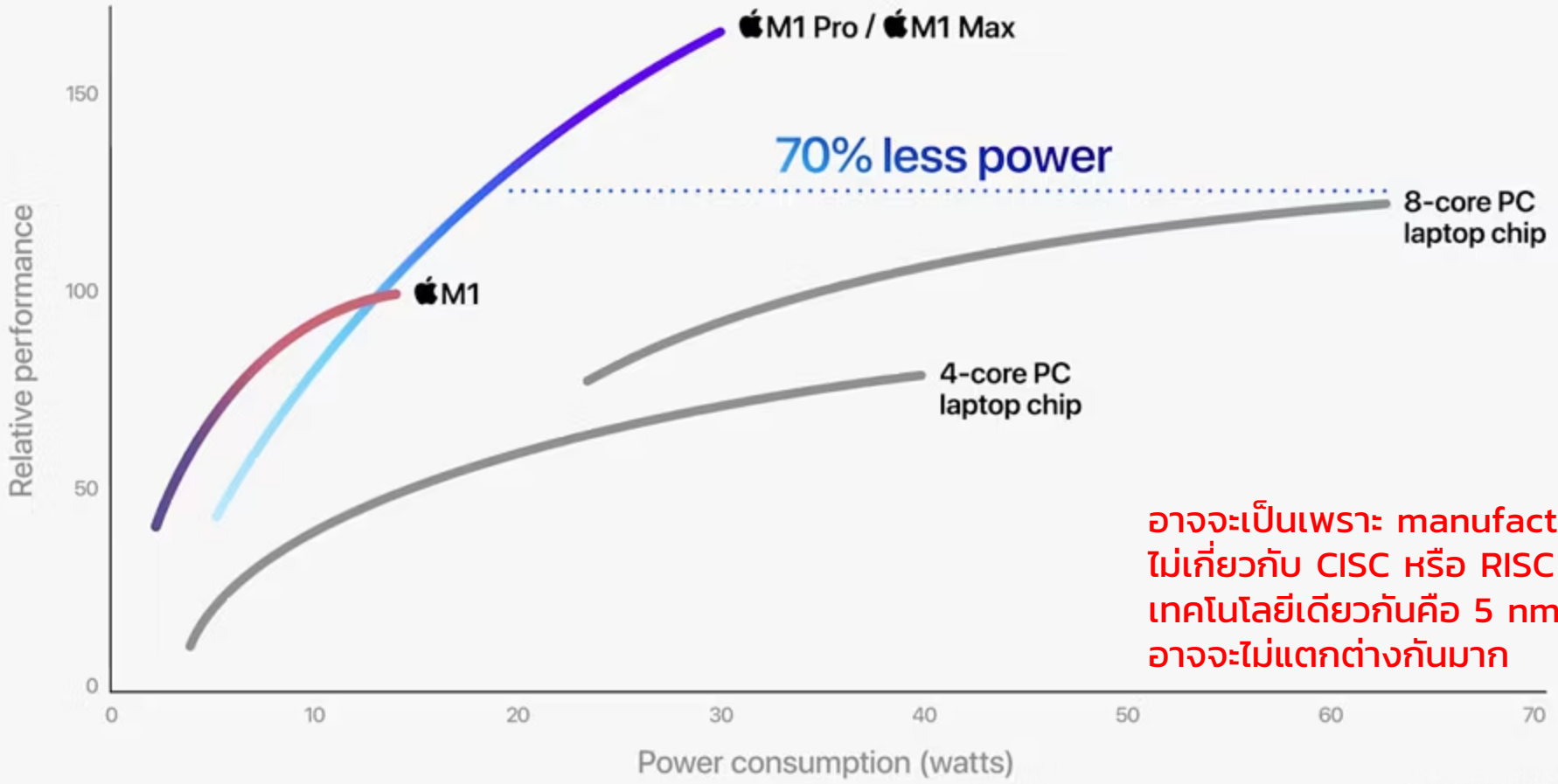
ARM (Advanced RISC Machines)

- Arm Ltd. Is a British company in Cambridge, England
- Arm develops the architectures and licenses them to other companies
- Low costs, low power consumption, and low heat generation
- In 2013, ARM-based chips are found in nearly 60 percent of the world's mobile devices

Market share ใน PC (พยากรณ์เมื่อ 2013)



CPU performance vs. power



อาจจะเป็นเพราะ manufacturing technology ไม่เกี่ยวกับ CISC หรือ RISC เช่น ถ้าผลิตที่เทคโนโลยีเดียวกันคือ 5 nm CISC กับ RISC อาจจะไม่แตกต่างกันมาก

4-core PC laptop performance data from testing MSI Prestige 14 EVO A11M-220
8-core PC laptop performance data from testing MSI GP66 Leopard 11UG-018

MIPS Architecture

- General-purpose registers (GPRs)
  - ขนาด 64 บิต จำนวน 32 ตัว
  - ใช้ตัวย่อ R0 – R31 เก็บจำนวนเต็ม
- Floating-point registers (FPRs)
  - ขนาด 64 บิต จำนวน 32 ตัว
  - ใช้ตัวย่อ F0 – F31 เก็บจำนวนทศนิยม
    - Single-precision value (32-bit)
    - Double-precision value (64-bit)
- มี instruction สำหรับ single precision และ double precision แยกกัน
- มี instruction สำหรับย้ายค่าระหว่าง GPR และ FPR
- R0 เป็นค่าคงที่ R0 = 0 เสมอ เพราะค่าศูนย์มักจะต้องใช้บ่อย ๆ

Addressing modes ของ MIPS

Addressing mode	Example instruction	Meaning	When used
Register	`Add R4,R3`	`Regs[R4] ← Regs[R4] + Regs[R3]`	When a value is in a register.
Immediate	`Add R4,#3`	`Regs[R4] ← Regs[R4] + 3`	For constants.
Displacement	`Add R4,100(R1)`	`Regs[R4] ← Regs[R4] + Mem[100 + Regs[R1]]`	Accessing local variables (+ simulates register indirect, direct addressing modes).
Register indirect	`Add R4,(R1)`	`Regs[R4] ← Regs[R4] + Mem[Regs[R1]]`	Accessing using a pointer or a computed address.
Indexed	`Add R3,(R1 + R2)`	`Regs[R3] ← Regs[R3] + Mem[Regs[R1] + Regs[R2]]`	Sometimes useful in array addressing: R1 = base of array; R2 = index amount.
Direct or absolute	`Add R1,(1001)`	`Regs[R1] ← Regs[R1] + Mem[1001]`	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	`Add R1,@(R3)`	`Regs[R1] ← Regs[R1] + Mem[Mem[Regs[R3]]]`	If R3 is the address of a pointer *p*, then mode yields **p*.

Autoincrement	`Add R1,(R2)+`	`Regs[R1] ← Regs[R1] + Mem[Regs[R2]]` `Regs[R2] ← Regs[R2] + d`	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, *d*.
Autodecrement	`Add R1, –(R2)`	`Regs[R2] ← Regs[R2] – d` `Regs[R1] ← Regs[R1] + Mem[Regs[R2]]`	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	`Add R1,100(R2)[R3]`	`Regs[R1] ← Regs[R1] + Mem[100 + Regs[R2] + Regs[R3] * d]`	Used to index arrays. May be applied to any indexed addressing mode in some computers.

Example instruction	Instruction name	Meaning
LD R1,30(R2)	Load double word	Regs[R1] $\leftarrow_{64}$ Mem[30+Regs[R2]]
LD R1,1000(R0)	Load double word	Regs[R1] $\leftarrow_{64}$ Mem[1000+0]
LW R1,60(R2)	Load word	Regs[R1] $\leftarrow_{64}$ (Mem[60+Regs[R2]]$_0$)^{32} ## Mem[60+Regs[R2]]
LB R1,40(R3)	Load byte	Regs[R1] $\leftarrow_{64}$ (Mem[40+Regs[R3]]$_0$)^{56} ## Mem[40+Regs[R3]]
LBU R1,40(R3)	Load byte unsigned	Regs[R1] $\leftarrow_{64}$ 0^{56} ## Mem[40+Regs[R3]]
LH R1,40(R3)	Load half word	Regs[R1] $\leftarrow_{64}$ (Mem[40+Regs[R3]]$_0$)^{48} ## Mem[40+Regs[R3]] ## Mem[41+Regs[R3]]
L.S F0,50(R3)	Load FP single	Regs[F0] $\leftarrow_{64}$ Mem[50+Regs[R3]] ## 0^{32}
L.D F0,50(R2)	Load FP double	Regs[F0] $\leftarrow_{64}$ Mem[50+Regs[R2]]
SD R3,500(R4)	Store double word	Mem[500+Regs[R4]] $\leftarrow_{64}$ Regs[R3]
SW R3,500(R4)	Store word	Mem[500+Regs[R4]] $\leftarrow_{32}$ Regs[R3]$_{32..63}$
S.S F0,40(R3)	Store FP single	Mem[40+Regs[R3]] $\leftarrow_{32}$ Regs[F0]$_{0..31}$
S.D F0,40(R3)	Store FP double	Mem[40+Regs[R3]] $\leftarrow_{64}$ Regs[F0]
SH R3,502(R2)	Store half	Mem[502+Regs[R2]] $\leftarrow_{16}$ Regs[R3]$_{48..63}$
SB R2,41(R3)	Store byte	Mem[41+Regs[R3]] $\leftarrow_{8}$ Regs[R2]$_{56..63}$

Figure A.23 The load and store instructions in MIPS. All use a single addressing mode and require that the memory value be aligned. Of course, both loads and stores are available for all the data types shown.

- A subscript is appended to the symbol $\leftarrow$ whenever the length of the datum being transferred might not be clear. Thus, $\leftarrow_n$ means transfer an n-bit quantity. We use $x, y \leftarrow z$ to indicate that z should be transferred to x and y.
- A subscript is used to indicate selection of a bit from a field. Bits are labeled from the most-significant bit starting at 0. The subscript may be a single digit (e.g., Regs[R4]$_0$ yields the sign bit of R4) or a subrange (e.g., Regs[R3]$_{56..63}$ yields the least-significant byte of R3).
- The variable Mem, used as an array that stands for main memory, is indexed by a byte address and may transfer any number of bytes.
- A superscript is used to replicate a field (e.g., 0^{48} yields a field of zeros of length 48 bits).
- The symbol ## is used to concatenate two fields and may appear on either side of a data transfer.

Example instruction	Instruction name	Meaning
DADDU R1,R2,R3	Add unsigned	Regs[R1]← Regs[R2]+Regs[R3]
DADDIU R1,R2,#3	Add immediate unsigned	Regs[R1]← Regs[R2]+3
LUI R1,#42	Load upper immediate	Regs[R1]← 0^{32}##42##0^{16}
DSLL R1,R2,#5	Shift left logical	Regs[R1]← Regs[R2]<<5
SLT R1,R2,R3	Set less than	if (Regs[R2]<Regs[R3]) Regs[R1] ← 1 else Regs[R1]←0

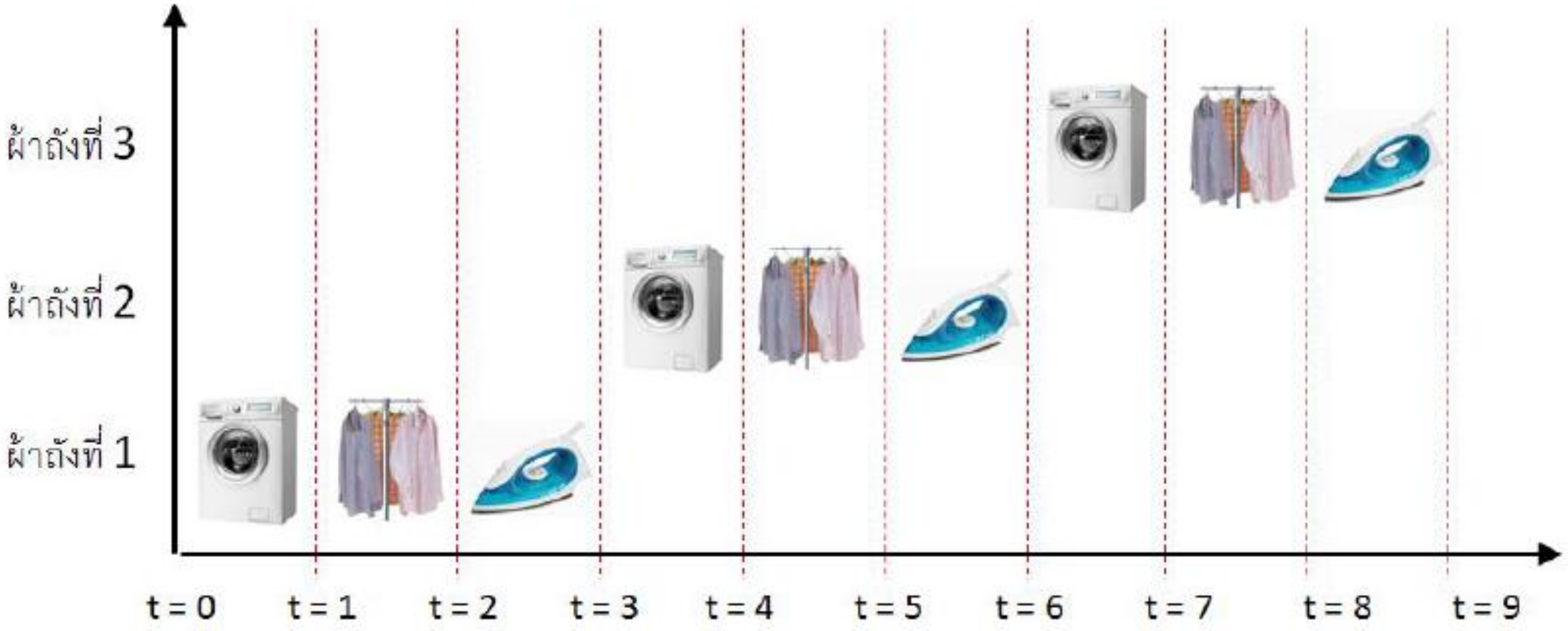
Example instruction		Instruction name	Meaning
J	name	Jump	$PC_{36..63} \leftarrow$ name
JAL	name	Jump and link	Regs[R31] $\leftarrow$ PC+8; $PC_{36..63} \leftarrow$ name; ((PC + 4) -2^{27}) $\leq$ name < ((PC+4) $+2^{27}$)
JALR	R2	Jump and link register	Regs[R31] $\leftarrow$ PC+8; PC $\leftarrow$ Regs[R2]
JR	R3	Jump register	PC $\leftarrow$ Regs[R3]
BEQZ	R4,name	Branch equal zero	if (Regs[R4]==0) PC $\leftarrow$ name; ((PC+4) -2^{17}) $\leq$ name < ((PC+4) $+2^{17}$)
BNE	R3,R4,name	Branch not equal ~~zero~~	if (Regs[R3] != Regs[R4]) PC $\leftarrow$ name; ((PC+4) -2^{17}) $\leq$ name < ((PC+4) $+2^{17}$)
MOVZ	R1,R2,R3	Conditional move if zero	if (Regs[R3]==0) Regs[R1] $\leftarrow$ Regs[R2]

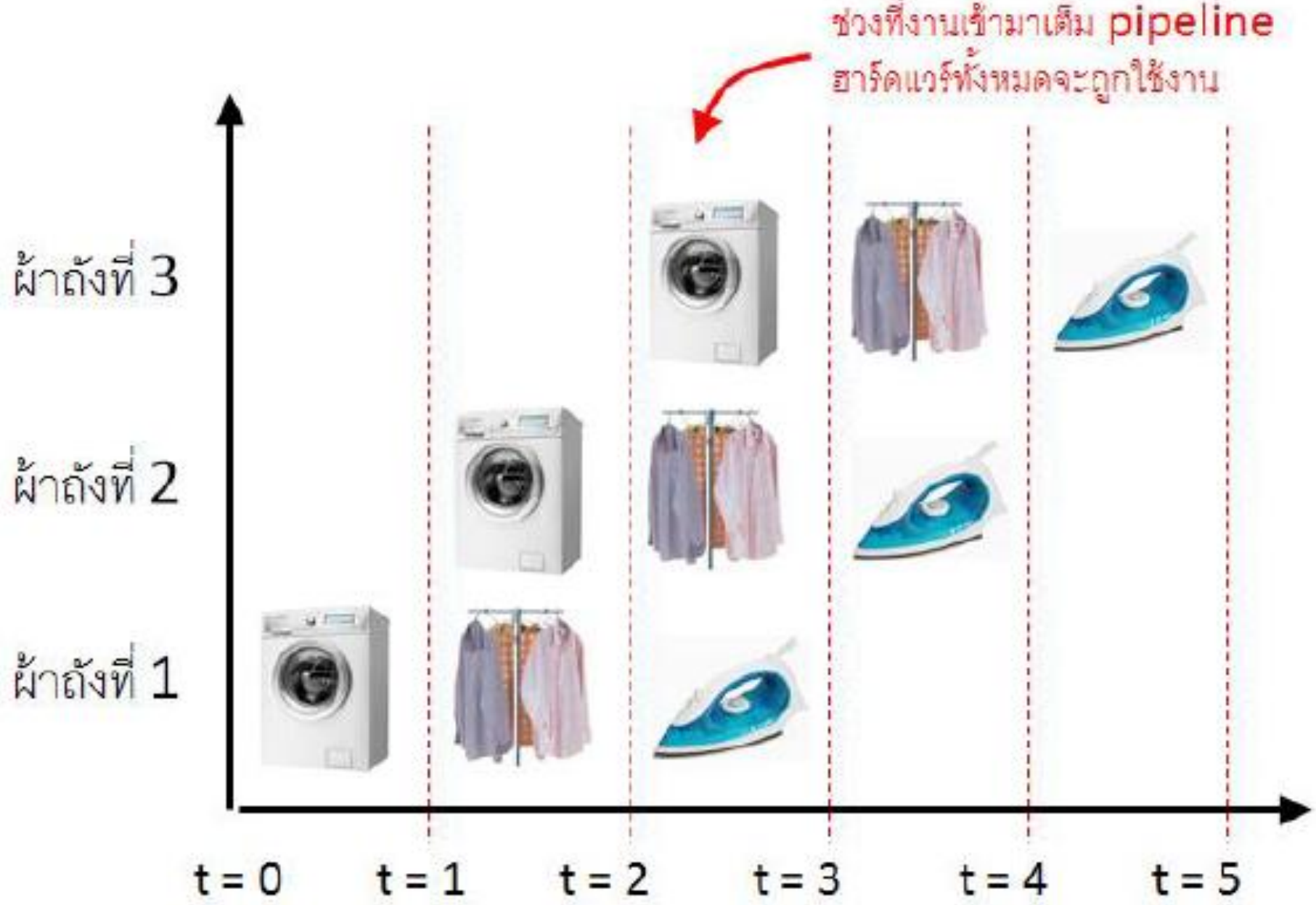
ใช้ค่าใน register ไม่ใช้ค่าที่เขียน register ครั้งสุดท้ายเหมือน LC3

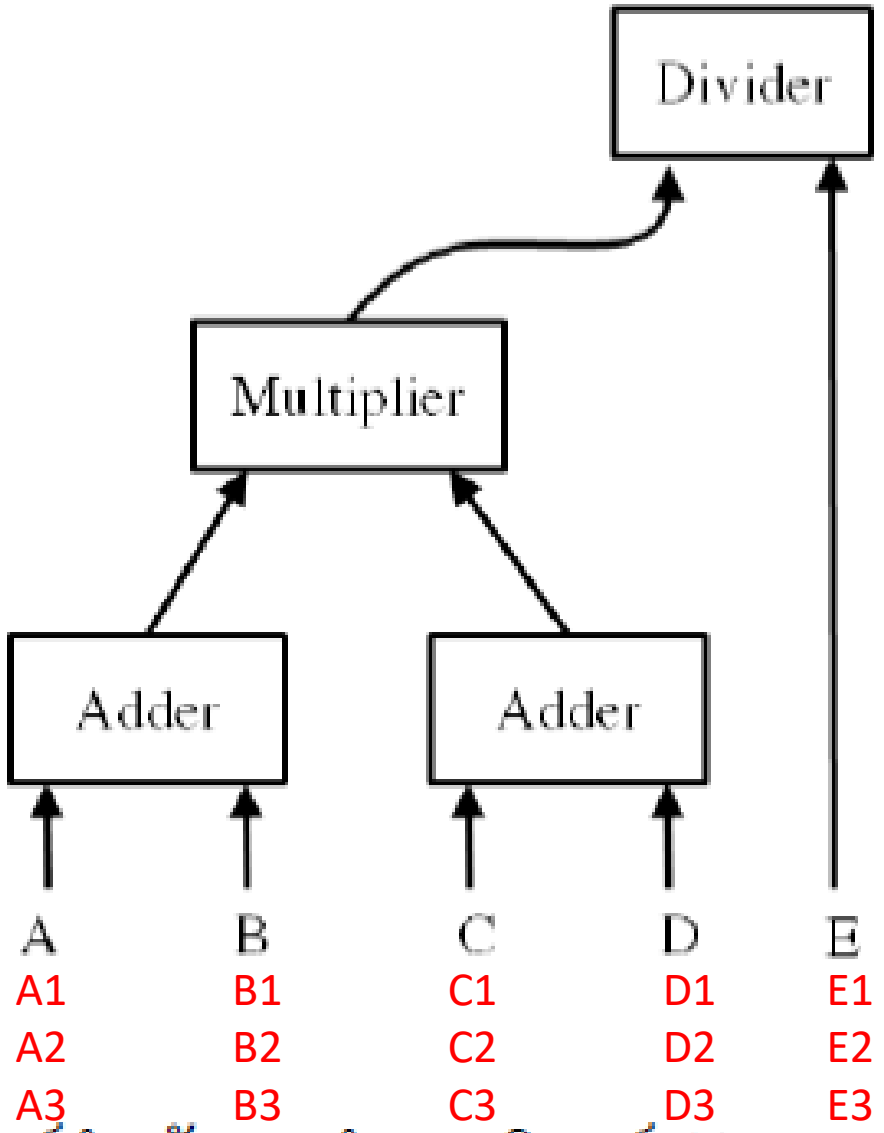
Ford Assembly Line

https://www.youtube.com/watch?v=cTZ3rJHHSik

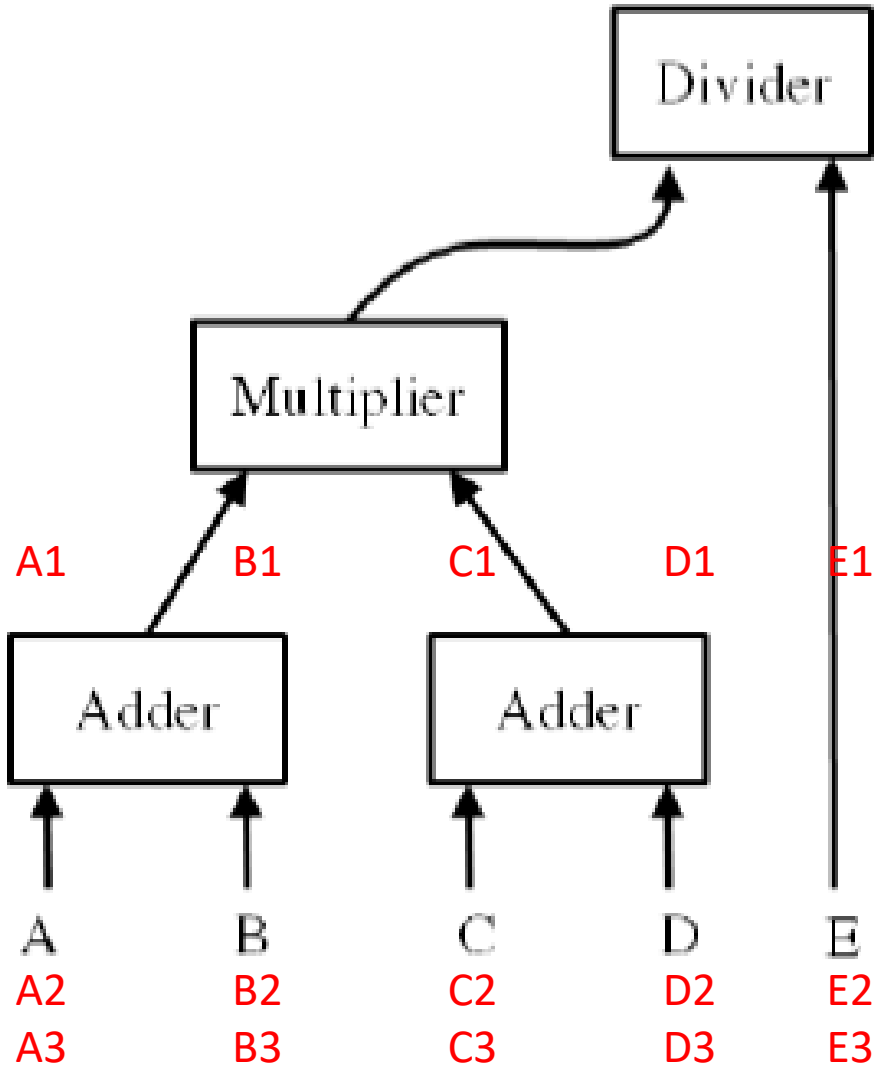
1 instruction = ผ้า 1 ถัง
Processor = ร้านซักรีด



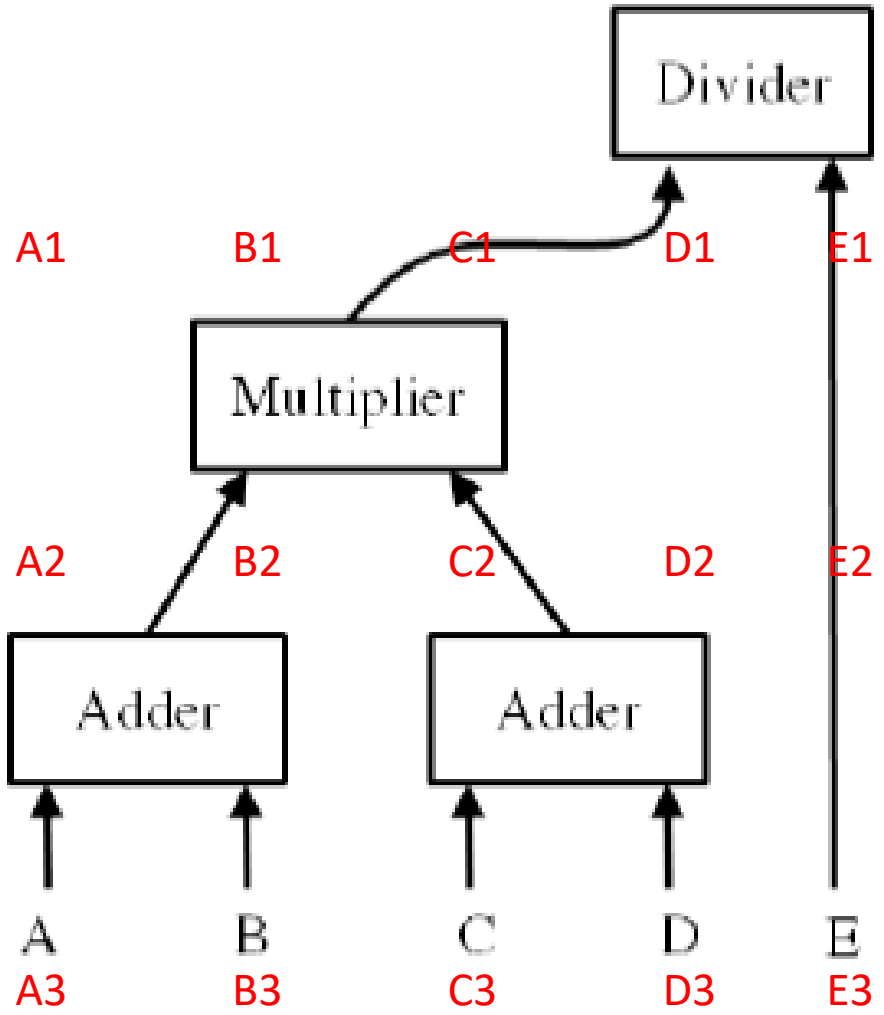

ช่วงที่งานเข้ามาเต็ม pipeline
ฮาร์ดแวร์ทั้งหมดจะถูกใช้งาน
ผ้าถังที่ 3
ผ้าถังที่ 2
ผ้าถังที่ 1
t = 0
t = 1
t = 2
t = 3
t = 4
t = 5



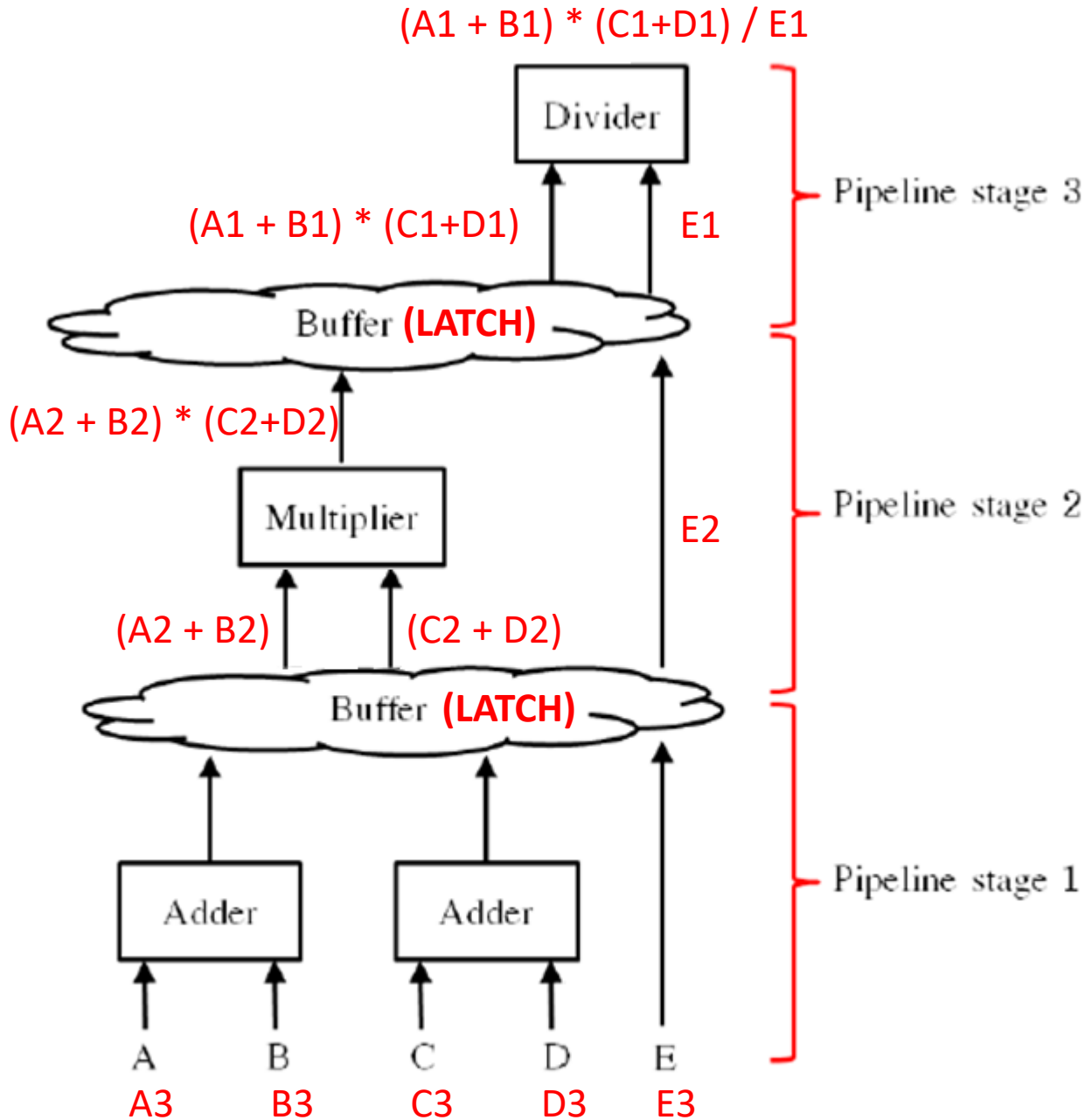
รูปที่ 13.5 ฮาร์ดแวร์สำหรับการคำนวณนิพจน์ ((A + B) x (C + D)) / E



รูปที่ 13.5 ฮาร์ดแวร์สำหรับการคำนวณนิพจน์ ((A + B) x (C + D)) / E



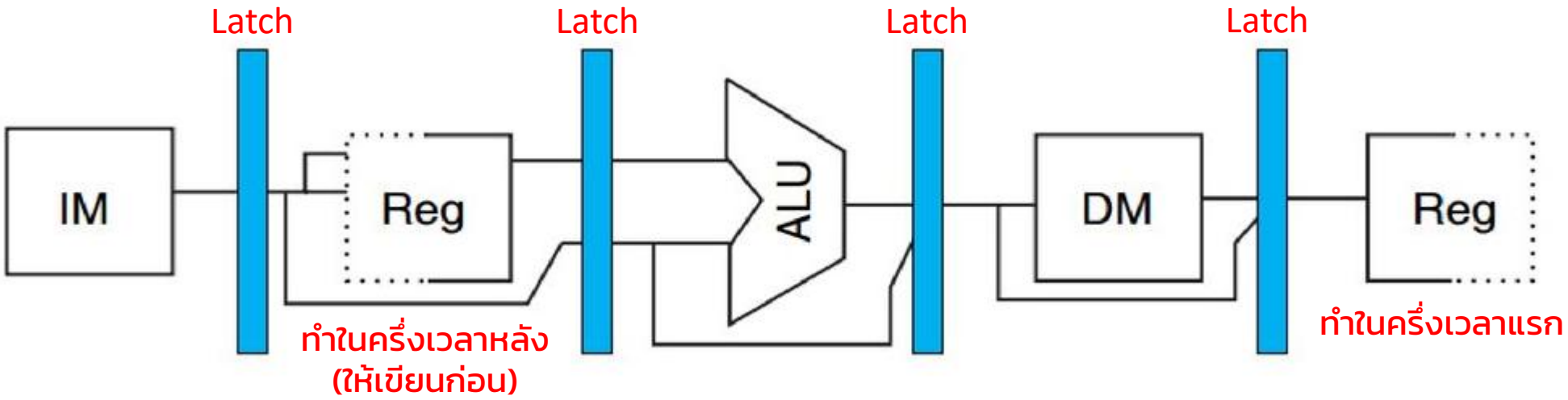
รูปที่ 13.5 ฮาร์ดแวร์สำหรับการคำนวณนิพจน์ ((A + B) x (C + D)) / E



รูปที่ 13.6 ฮาร์ดแวร์สำหรับการคำนวณนิพจน์ ((A + B) x (C + D)) / E แบบสายท่อ

Hardware มี IM, Reg, ALU, DM, Reg
Pipeline มี IF, ID, EX, MEM, WB

IM = instruction memory, DM = data memory



1. *Instruction fetch cycle* (IF):

   Send the program counter (PC) to memory and fetch the current instruction from memory. Update the PC to the next sequential PC by adding 4 (since each instruction is 4 bytes) to the PC.

2. *Instruction decode/register fetch cycle* (ID):

Decode the instruction and read the registers corresponding to register source specifiers from the register file. Do the equality test on the registers as they are read, for a possible branch. Sign-extend the offset field of the instruction in case it is needed. Compute the possible branch target address by adding the sign-extended offset to the incremented PC. In an aggressive implementation, which we explore later, the branch can be completed at the end of this stage by storing the branch-target address into the PC, if the condition test yielded true.

Decoding is done in parallel with reading registers, which is possible because the register specifiers are at a fixed location in a RISC architecture.

This technique is known as *fixed-field decoding*. Note that we may read a register we don't use, which doesn't help but also doesn't hurt performance. (It does waste energy to read an unneeded register, and power-sensitive designs might avoid this.) Because the immediate portion of an instruction is also located in an identical place, the sign-extended immediate is also calculated during this cycle in case it is needed.

3. *Execution/effective address cycle* (EX):

The ALU operates on the operands prepared in the prior cycle, performing one of three functions depending on the instruction type.

- Memory reference—The ALU adds the base register and the offset to form the effective address.
- Register-Register ALU instruction—The ALU performs the operation specified by the ALU opcode on the values read from the register file.
- Register-Immediate ALU instruction—The ALU performs the operation specified by the ALU opcode on the first value read from the register file and the sign-extended immediate.

In a load-store architecture the effective address and execution cycles can be combined into a single clock cycle, since no instruction needs to simultaneously calculate a data address and perform an operation on the data.

4. *Memory access* (MEM):

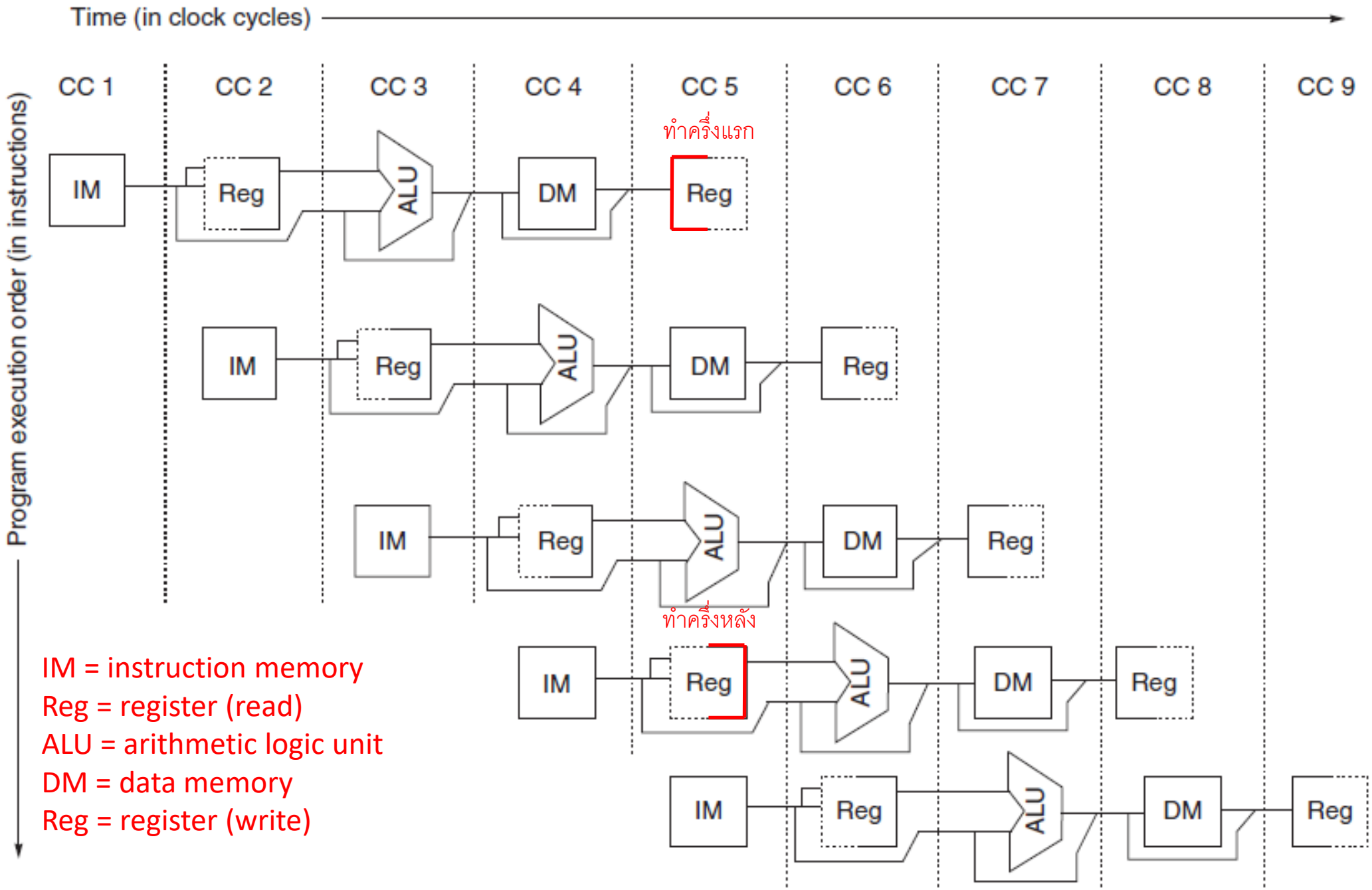
   If the instruction is a load, the memory does a read using the effective address computed in the previous cycle. If it is a store, then the memory writes the data from the second register read from the register file using the effective address.

5. *Write-back cycle* (WB):

   - Register-Register ALU instruction or load instruction:

   Write the result into the register file, whether it comes from the memory system (for a load) or from the ALU (for an ALU instruction).

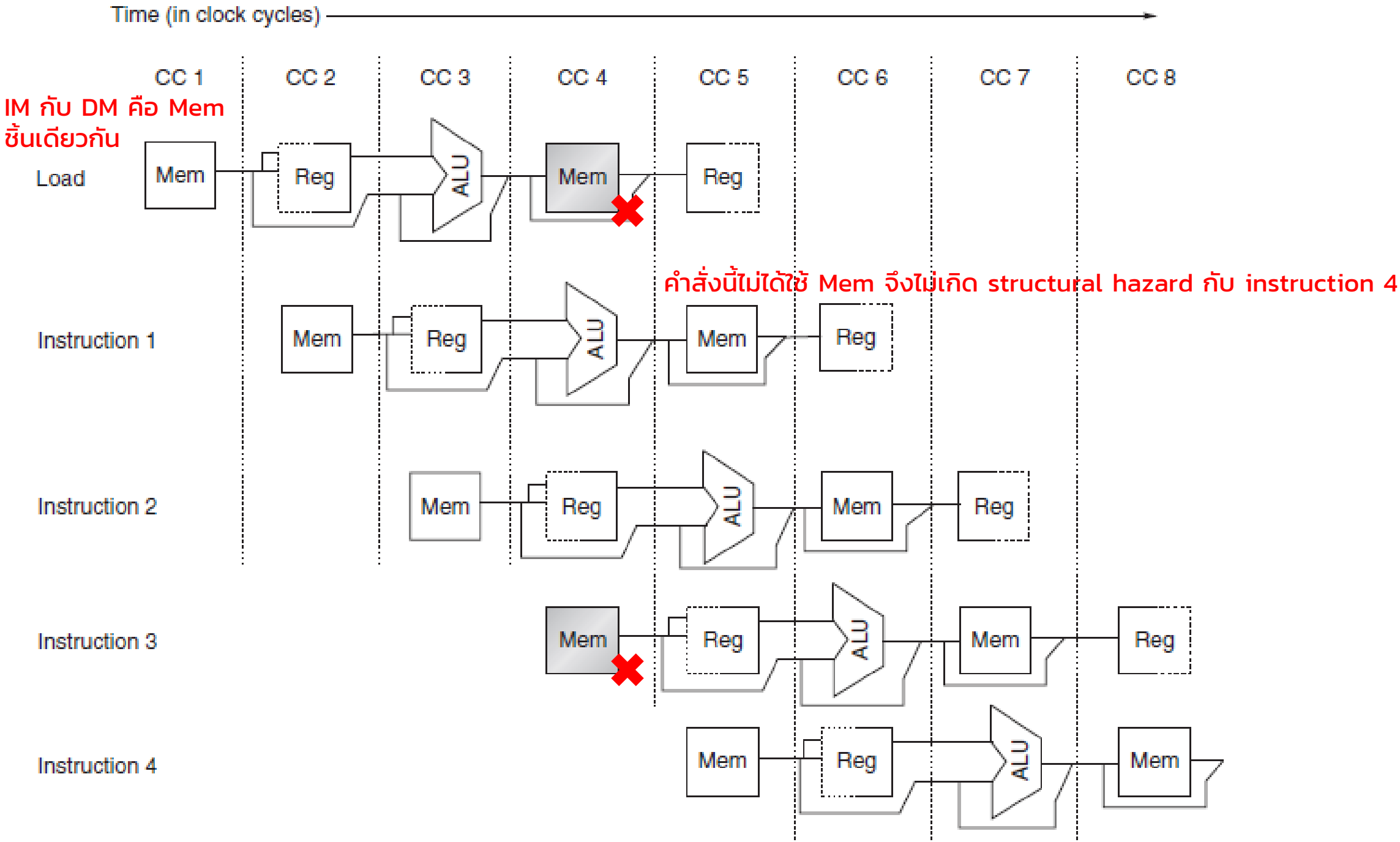
	Clock number								
Instruction number	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	ID	EX	MEM	WB			
Instruction $i + 2$			IF	ID	EX	MEM	WB		
Instruction $i + 3$				IF	ID	EX	MEM	WB	
Instruction $i + 4$					IF	ID	EX	MEM	WB


Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
Program execution order (in instructions)
IM
Reg
ALU
DM
Reg
ทำครึ่งแรก
ทำครึ่งหลัง
IM = instruction memory
Reg = register (read)
ALU = arithmetic logic unit
DM = data memory
Reg = register (write)

The Major Hurdle of Pipelining—Pipeline Hazards

There are situations, called *hazards*, that prevent the next instruction in the instruction stream from executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining. There are three classes of hazards:

1. *Structural hazards* arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution.
2. *Data hazards* arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
3. *Control hazards* arise from the pipelining of branches and other instructions that change the PC.


Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
IM กับ DM คือ Mem ชิ้นเดียวกัน
Load
Mem
Reg
ALU
Mem
Reg
คำสั่งนี้ไม่ได้ใช้ Mem จึงไม่เกิด structural hazard กับ instruction 4
Instruction 1
Mem
Reg
ALU
Mem
Reg
Instruction 2
Mem
Reg
ALU
Mem
Reg
Instruction 3
Mem
Reg
ALU
Mem
Reg
Instruction 4
Mem
Reg
ALU
Mem

	Clock cycle number									
Instruction	1	2	3	4	5	6	7	8	9	10
Load instruction	IF	ID	EX	MEM	WB					
Instruction $i + 1$		IF	ID	EX	MEM	WB				
Instruction $i + 2$			IF	ID	EX	MEM	WB			
Instruction $i + 3$				Stall	IF	ID	EX	MEM	WB	
Instruction $i + 4$						IF	ID	EX	MEM	WB
Instruction $i + 5$							IF	ID	EX	MEM
Instruction $i + 6$								IF	ID	EX

ถ้าไม่ stall ก็ต้องแทรกคำสั่ง NOP (no operation)

- แก้ปัญหาด้วยฮาร์ดแวร์ processor เช่น ตรวจพบ hazard แล้ว stall ด้วยตัวเอง Complex hardware, backward compatibility (ใช้ exe เดิม ไม่ต้อง compile ใหม่)
- แก้ปัญหาด้วยซอฟต์แวร์ compiler เช่น เติมคำสั่ง NOP ก่อนคำสั่งที่ต้อง stall Simple hardware, ไม่ backward compatibility (ใช้ exe เดิมไม่ได้ ต้อง compile ใหม่)

Data Hazards

A major effect of pipelining is to change the relative timing of instructions by overlapping their execution. This overlap introduces data and control hazards. Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on an unpipelined processor. Consider the pipelined execution of these instructions:

```
DADD    R1,R2,R3
DSUB    R4,R1,R5
AND     R6,R1,R7
OR      R8,R1,R9
XOR     R10,R1,R11
```

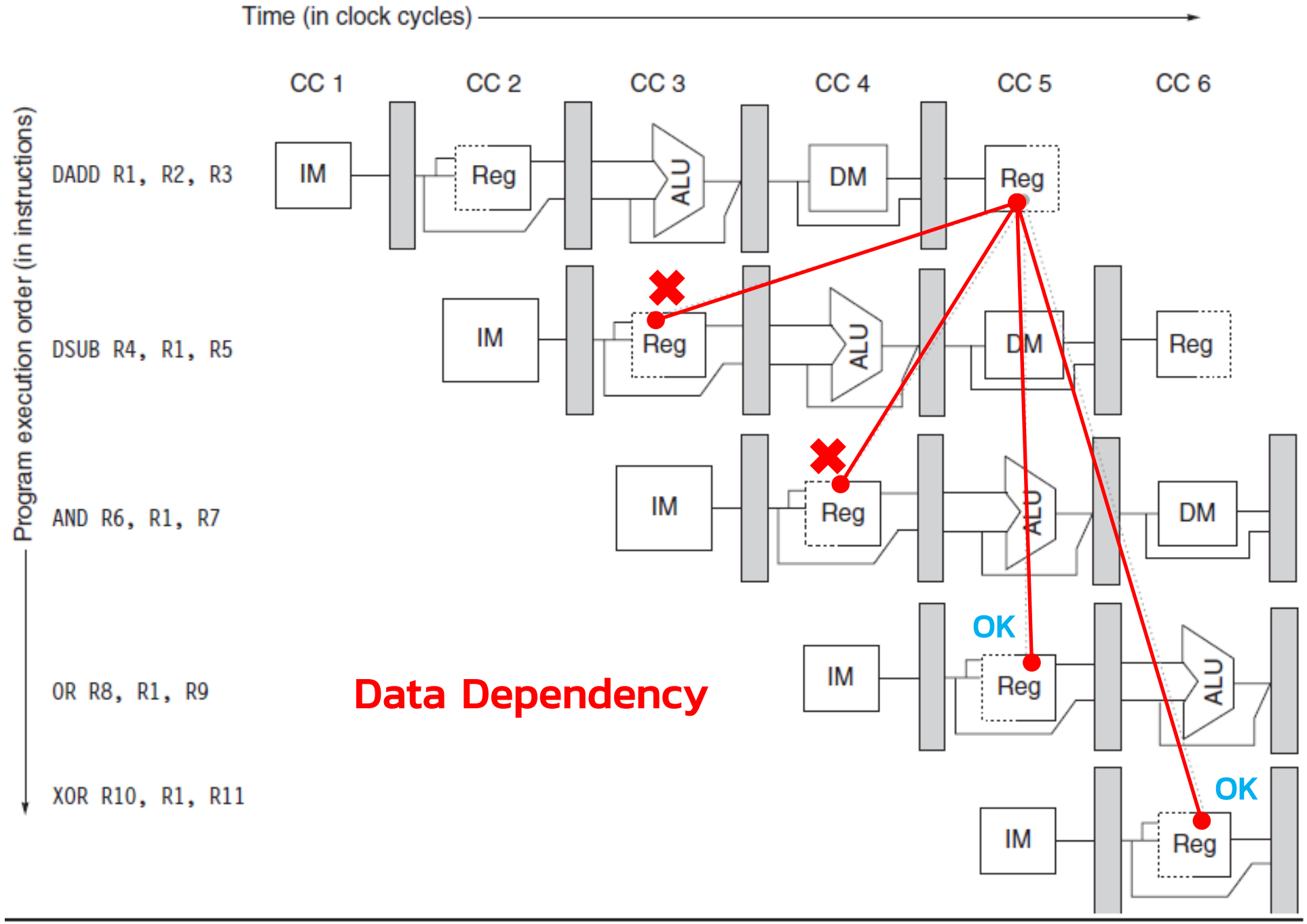
R1 = R2 + R3

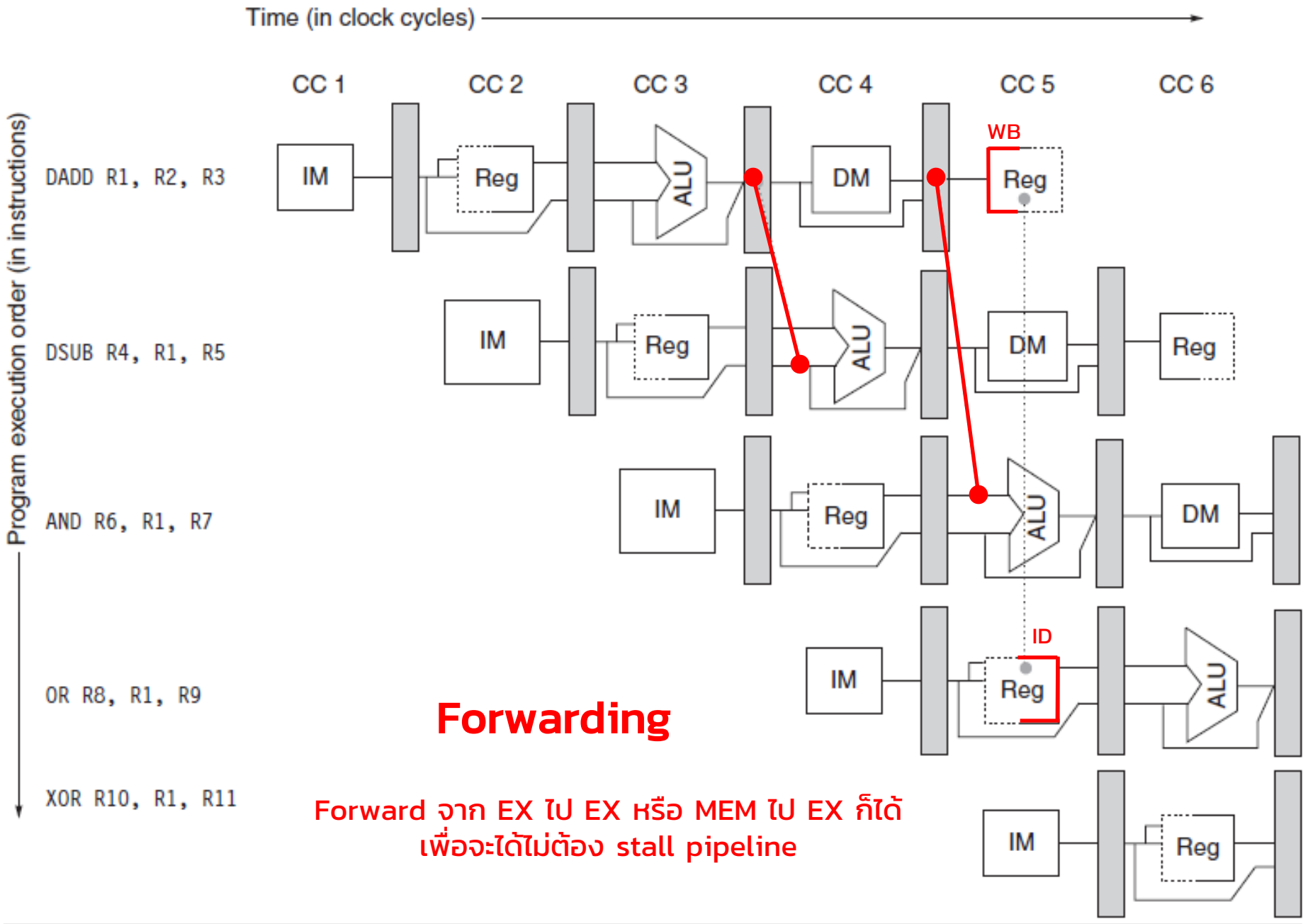
R4 = R1 – R5

R6 = R1 & R7

R8 = R1 | R9

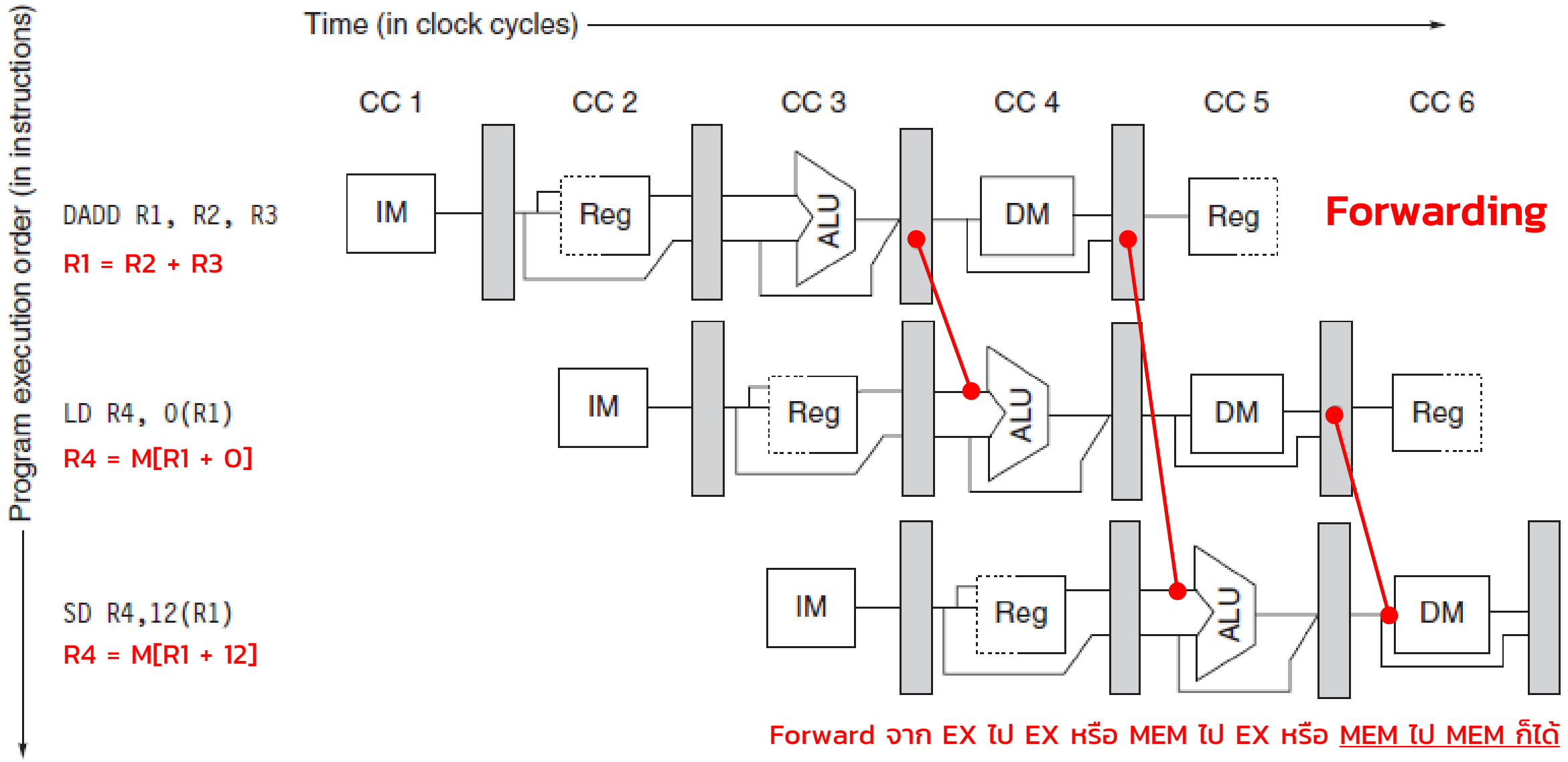
R10 = R1 ⊕ R11


Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
Program execution order (in instructions)
DADD R1, R2, R3
DSUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11
IM
Reg
ALU
DM
Reg
OK
OK
Data Dependency



Forwarding

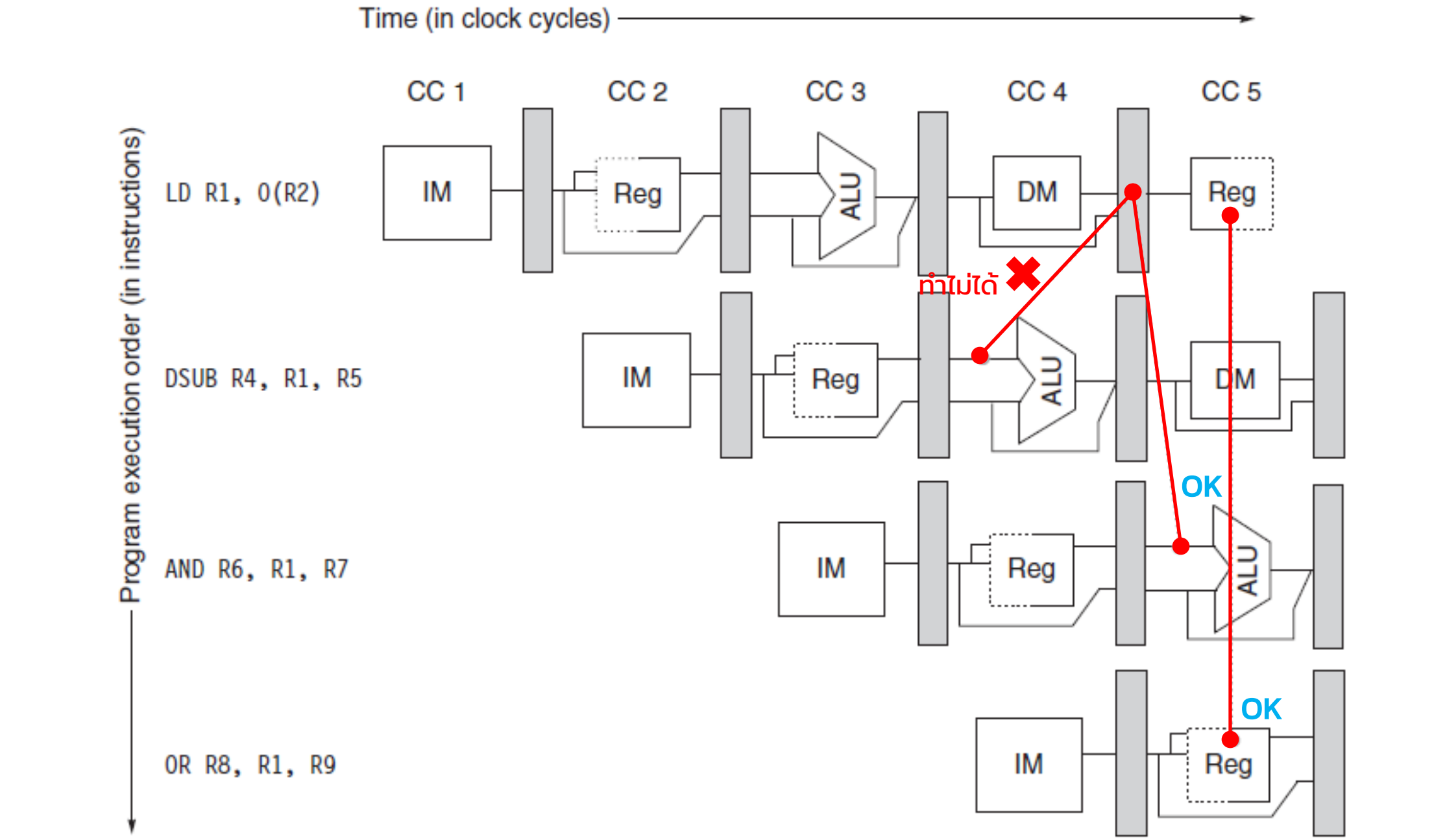
Forward จาก EX ไป EX หรือ MEM ไป EX ก็ได้
เพื่อจะได้ไม่ต้อง stall pipeline


Time (in clock cycles)
Program execution order (in instructions)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
DADD R1, R2, R3
R1 = R2 + R3
LD R4, 0(R1)
R4 = M[R1 + 0]
SD R4,12(R1)
R4 = M[R1 + 12]
IM
Reg
ALU
DM
Reg
Forwarding
Forward จาก EX ไป EX หรือ MEM ไป EX หรือ MEM ไป MEM ก็ได้

Data Hazards Requiring Stalls

Unfortunately, not all potential data hazards can be handled by bypassing. Consider the following sequence of instructions:

```
LD      R1,0(R2)
DSUB    R4,R1,R5
AND     R6,R1,R7
OR      R8,R1,R9
```


Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
Program execution order (in instructions)
LD R1, 0(R2)
DSUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
IM
Reg
ALU
DM
Reg
ทำไม่ได้
OK
OK

ถ้าไม่ stall ผลลัพธ์จะผิด

LD	R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB	R4,R1,R5		IF	ID	EX	MEM	WB			
AND	R6,R1,R7			IF	ID	EX	MEM	WB		
OR	R8,R1,R9				IF	ID	EX	MEM	WB	

stall & forward

LD	R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB	R4,R1,R5		IF	ID	stall	EX	MEM	WB		
AND	R6,R1,R7			IF	stall	ID	EX	MEM	WB	
OR	R8,R1,R9				stall	IF	ID	EX	MEM	WB

ถ้าไม่มี forwarding

LD	R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB	R4,R1,R5		IF	stall	stall	ID	EX	MEM	WB	

write R1 ในครั้งแรก

read R1 ในครั้งหลัง

Branch Hazards

Control hazards can cause a greater performance loss for our MIPS pipeline than do data hazards. When a branch is executed, it may or may not change the PC to something other than its current value plus 4. Recall that if a branch changes the PC to its target address, it is a *taken* branch; if it falls through, it is *not taken*, or *untaken*. If instruction *i* is a taken branch, then the PC is normally not changed until the end of ID, after the completion of the address calculation and comparison.

Figure C.11 shows that the simplest method of dealing with branches is to redo the fetch of the instruction following a branch, once we detect the branch during ID (when instructions are decoded). The first IF cycle is essentially a stall, because it never performs useful work. You may have noticed that if the branch is untaken, then the repetition of the IF stage is unnecessary since the correct instruction was indeed fetched. We will develop several schemes to take advantage of this fact shortly.

One stall cycle for every branch will yield a performance loss of 10% to 30% depending on the branch frequency, so we will examine some techniques to deal with this loss.

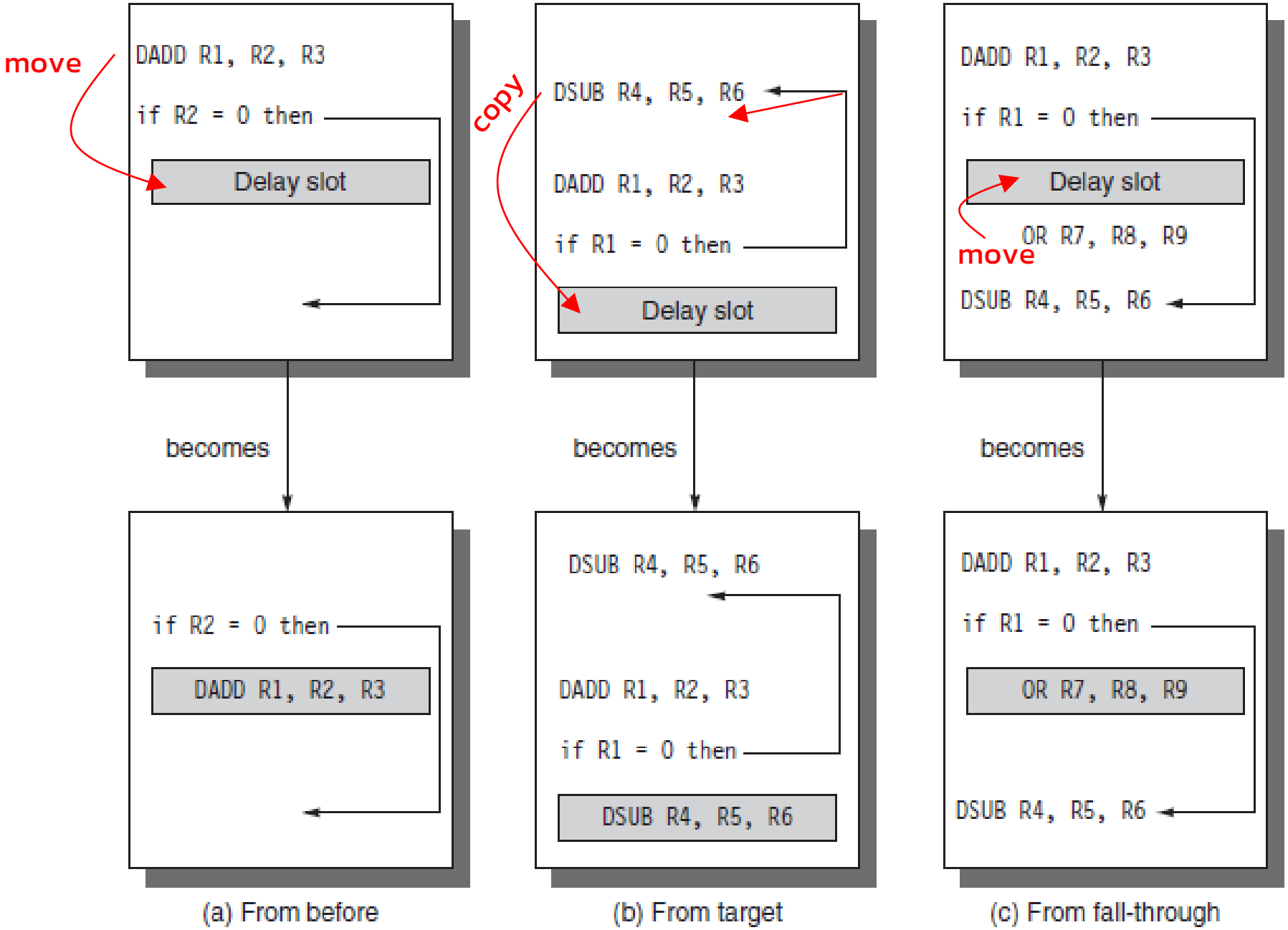
```
BEQZ R4,name
Branch equal zero
if (Regs[R4]==0) PC←name;
```

ชื่อ subroutine
หรือ mem. addr.

ต้องหลัง ID ถึงจะรู้ว่า branch taken หรือไม่

Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor + 1			stall เพราะไม่รู้ว่า	IF	ID	EX	MEM
Branch successor + 2			จะ taken หรือไม่		IF	ID	EX

Stall หรือ
ใส่ NOP (delay slot)



คำสั่งถัดจาก branch เรียกว่า "Delay Slot"

คำสั่งใน delay slot จะถูก executed เสมอไม่ว่า branch จะ taken หรือไม่

Situation	Example code sequence	Action
No dependence	LD R1,45(R2) DADD R5,R6,R7 DSUB R8,R6,R7 OR R9,R6,R7	No hazard possible because no dependence exists on R1 in the immediately following three instructions.
Dependence requiring stall	LD R1,45(R2) DADD R5,R1,R7 DSUB R8,R6,R7 OR R9,R6,R7	Comparators detect the use of R1 in the DADD and stall the DADD (and DSUB and OR) before the DADD begins EX.
Dependence overcome by forwarding	LD R1,45(R2) DADD R5,R6,R7 DSUB R8,R1,R7 OR R9,R6,R7	Comparators detect use of R1 in DSUB and forward result of load to ALU in time for DSUB to begin EX.
Dependence with accesses in order	LD R1,45(R2) DADD R5,R6,R7 DSUB R8,R6,R7 OR R9,R1,R7	No action required because the read of R1 by OR occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half. WB ทำครึ่งแรก ID ทำครึ่งหลัง

Execution Out of Order

ถ้าทำหลาย ๆ instruction คู่ขนานไปพร้อม ๆ กันได้ เรียกว่า Superscalar Processor
ต้องมีฮาร์ดแวร์สำหรับทำ IF, ID, EX, MEM, WB หลายชุด

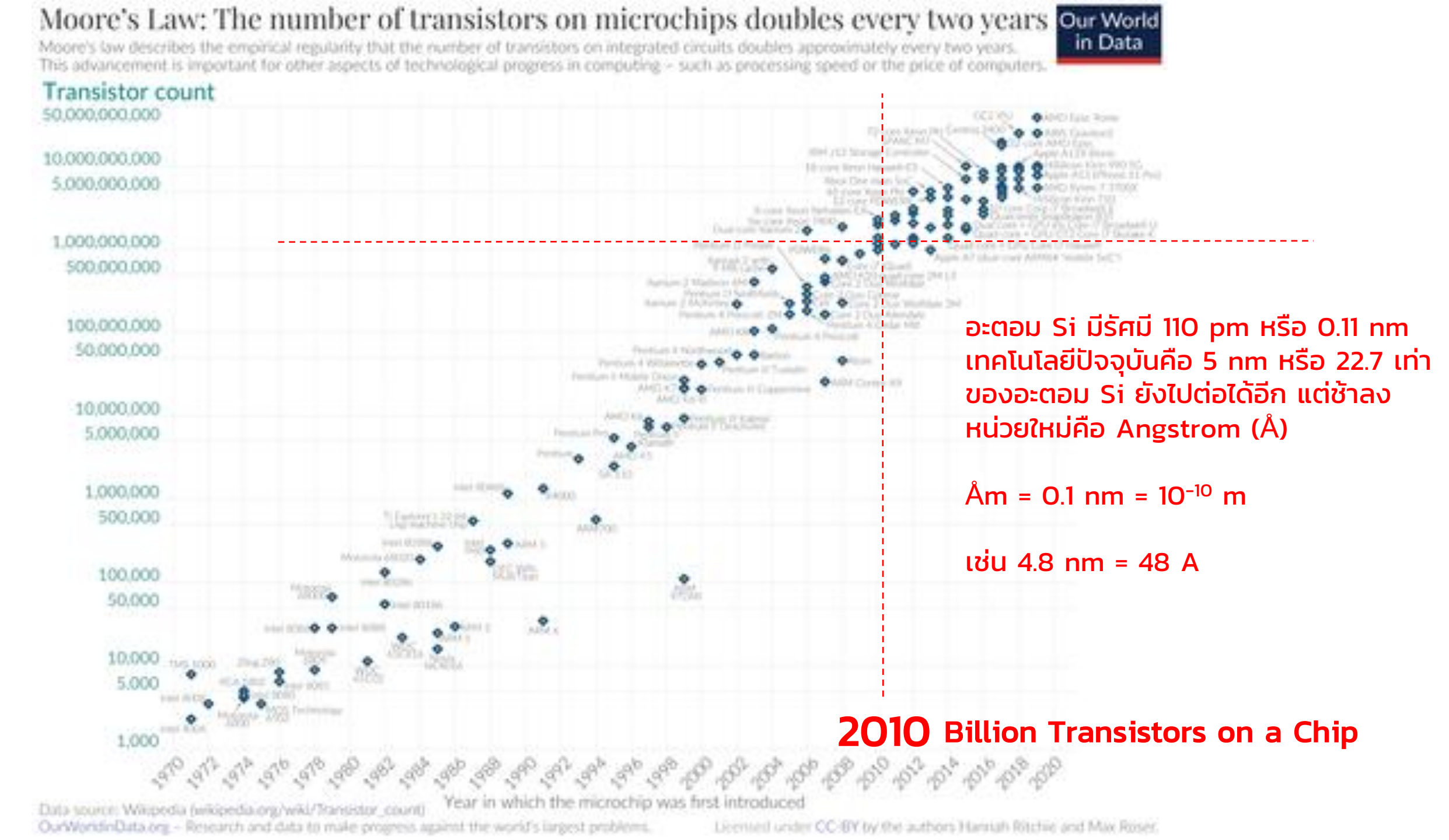
Pipeline

คำสั่งจะไม่วิ่งแซงกัน คำสั่งที่มาทีหลัง
จะเสร็จหลังคำสั่งที่มาก่อน

Superscalar

คำสั่งวิ่งแซงกันได้ คำสั่งที่มาทีหลัง
อาจจะทำเสร็จก่อน

คำสั่งที่มาก่อนเป็นคำสั่งที่ใช้เวลานาน เช่น
การคูณเลขทศนิยม ใช้เวลามากกว่าการบวก
เลขจำนวนเต็ม



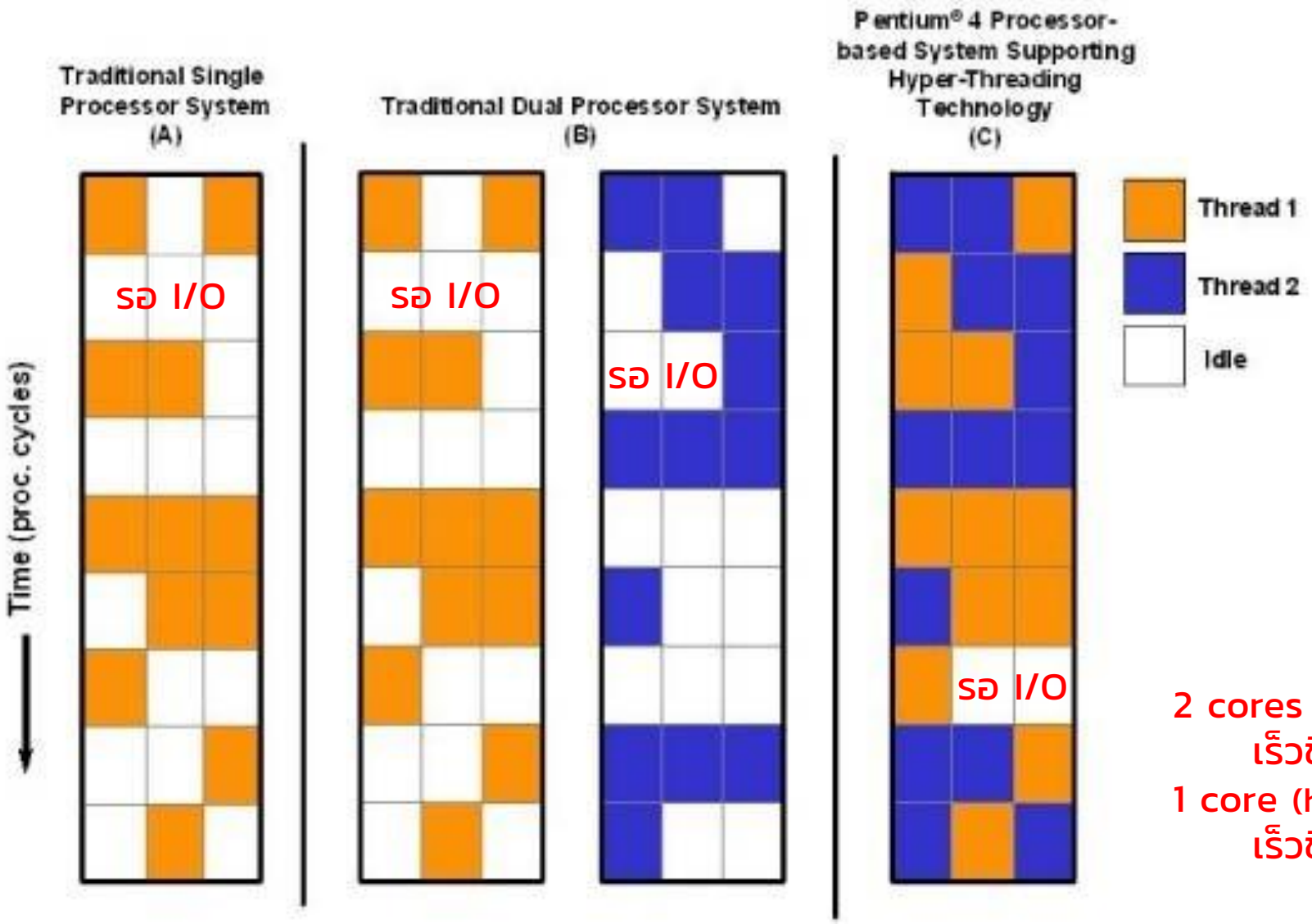
อะตอม Si มีรัศมี 110 pm หรือ 0.11 nm
เทคโนโลยีปัจจุบันคือ 5 nm หรือ 22.7 เท่า
ของอะตอม Si ยังไปต่อได้อีก แต่ช้าลง
หน่วยใหม่คือ Angstrom (Å)

Åm = 0.1 nm = 10^{-10} m

เช่น 4.8 nm = 48 A

2010 Billion Transistors on a Chip

Multi-Core (B) vs. Hyper-Threading (C)



1 physical processor (core)

Registers 1 ชุด

PC PC

1 logical processor

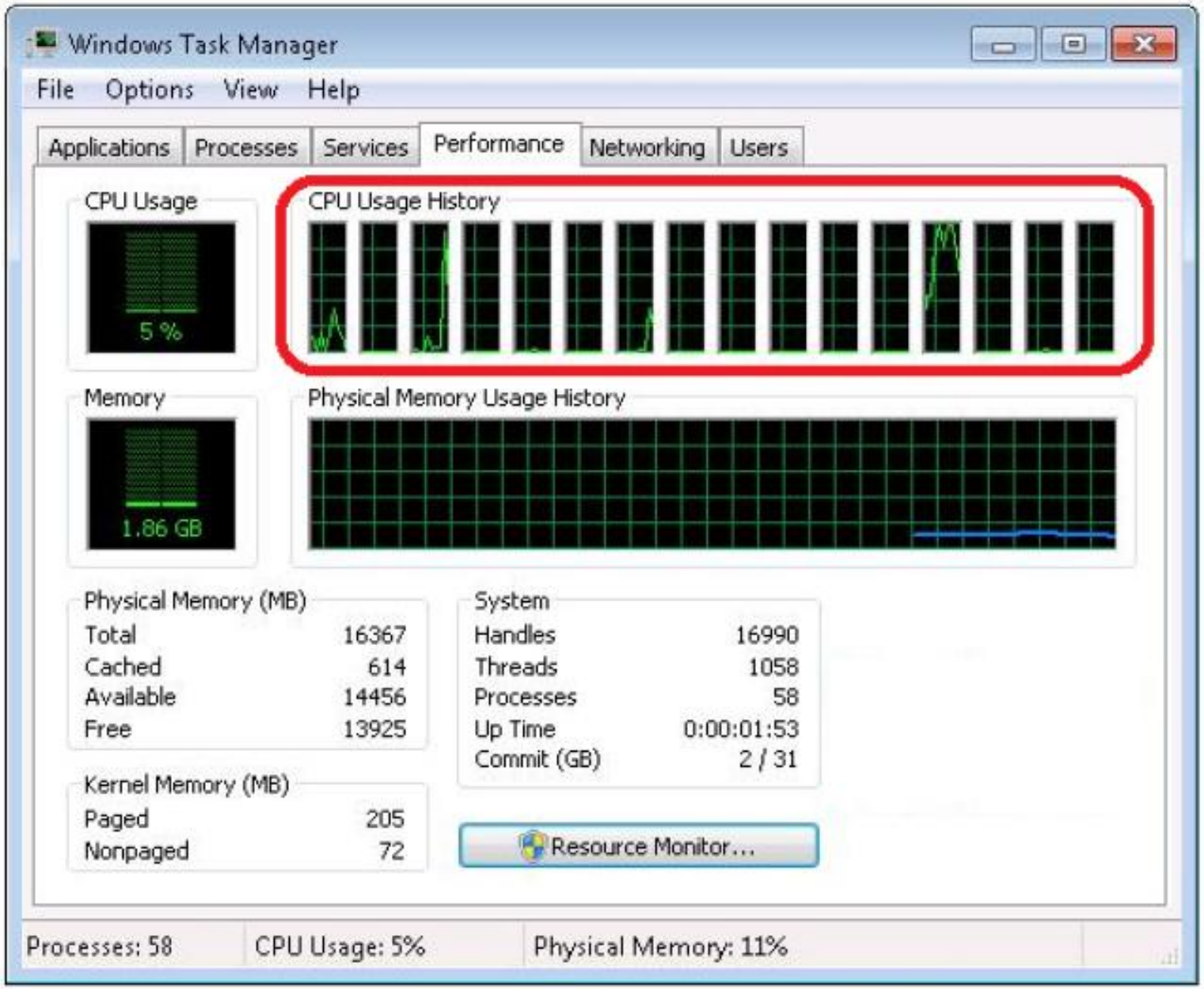
1 logical processor

HW ที่ช่วย switch ระหว่างโปรแกรม หรือ thread ใน 1 clock cycle

Hyper-threading ปกติจะทำแค่ 2 ชุด

2 cores (no hyper-threading) ได้สมรรถนะดีกว่า เร็วขึ้น 2 เท่า เทียบกับ 1 core

1 core (hyper-threading, 2 threads) เร็วขึ้น 30% เทียบกับ 1 thread



OS เห็น 16 logical processors
(8 cores + hyper-threading)

OS สั่ง execute โปรแกรมได้ 16 โปรแกรม
พร้อมกัน แต่ทำพร้อมกันได้แค่ทีละ 8 โปรแกรม

ใน BIOS เราสามารถ disable hyper-threading
ได้ OS จะเห็นแค่ 8 processors