Chapter 12
CPU Structure and Function

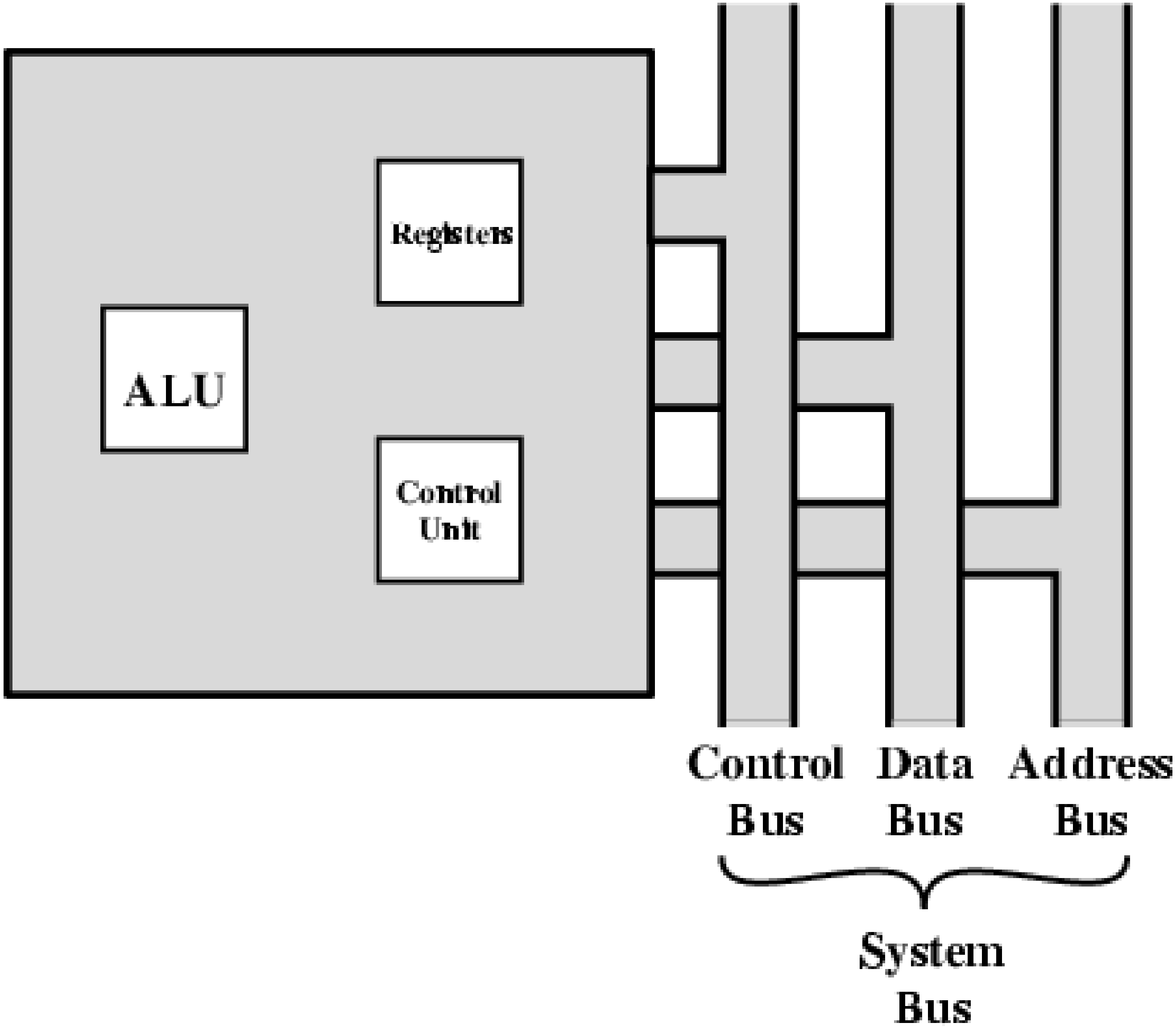
2020.6
Howon Kim

- 정보보호 및 지능형 IoT연구실 - http://infosec.pusan.ac.kr
- 부산대 지능형융합보안대학원 - http://aisec.pusan.ac.kr

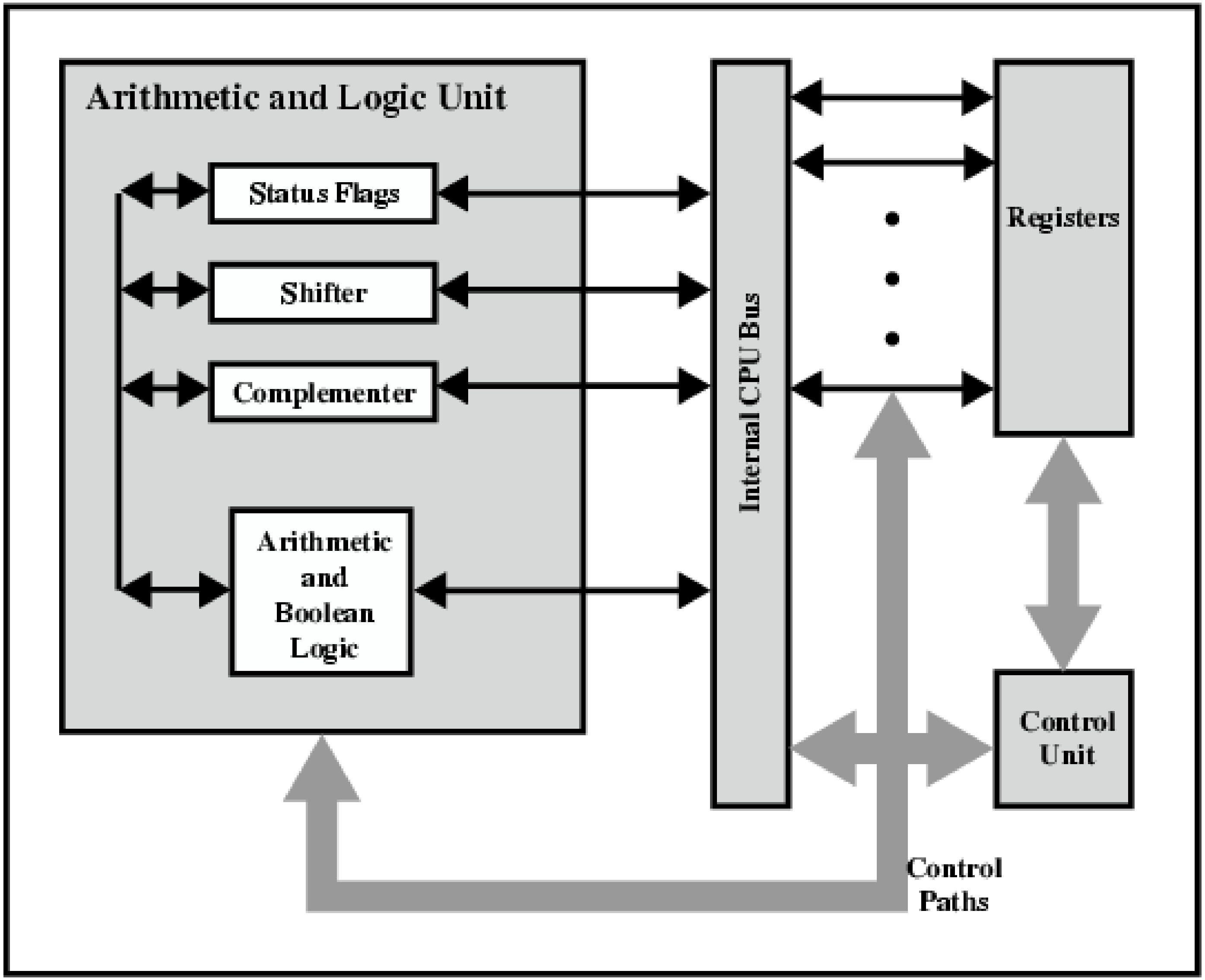
CPU Structure

- CPU must:
  - Fetch instructions
  - Interpret instructions
  - Fetch data
  - Process data
  - Write data
- Major components of CPU
  - ALU
  - Control unit
  - Register

CPU With Systems Bus



CPU Internal Structure



Registers

- CPU must have some working space (temporary storage), which called registers
- Number and function vary between processor designs
- One of the major design decisions
- Top level of memory hierarchy
    - —Small set of high speed storage

Registers

- Two types of registers
  - User visible registers:
    - may be referenced by assembly-level instruction
  - Control and status:
    - used by control unit to control CPU operations and by OS programs

User Visible Registers

- General Purpose
- Data
- Address
- Condition Codes

General Purpose Registers (1)

- May be used for true general purpose
  - E.g., PDP-11: R0 ~ R7
- May be used for restricted purpose
  - E.g., stack pointer, floating point registers
- May be used for data or addressing
  - Data
    - Accumulator, Motorola 68000: D0 ~ D7
  - Addressing
    - Segment , stack pointer, Motorola 68000: A0 ~ A7

General Purpose Registers (2)

- General purpose vs. Special purpose
- Make them general purpose
  - —Increase flexibility and programmer options
  - —Increase instruction size & complexity
- Make them specialized
  - —Smaller (faster) instructions
  - —Less flexibility
- Which is better?
  - —No final and best answer
  - —Trend toward specialized

General Purpose Registers (3)

- How many?
- Between 8 ~ 32
- Fewer → then, more memory references
- Also, more registers ***do not noticeably*** reduce memory references
  - However, a new approach, which finds advantage in the use of hundred of registers, is exhibited in some RISC !

General Purpose Registers (4)

- How big ?
- Large enough to hold full address
- Large enough to hold full word
- Often possible to combine two data registers
  - That is, two registers to hold one long integer
    - In C programming
    - long int a; // 64 bits

Control & Status Registers

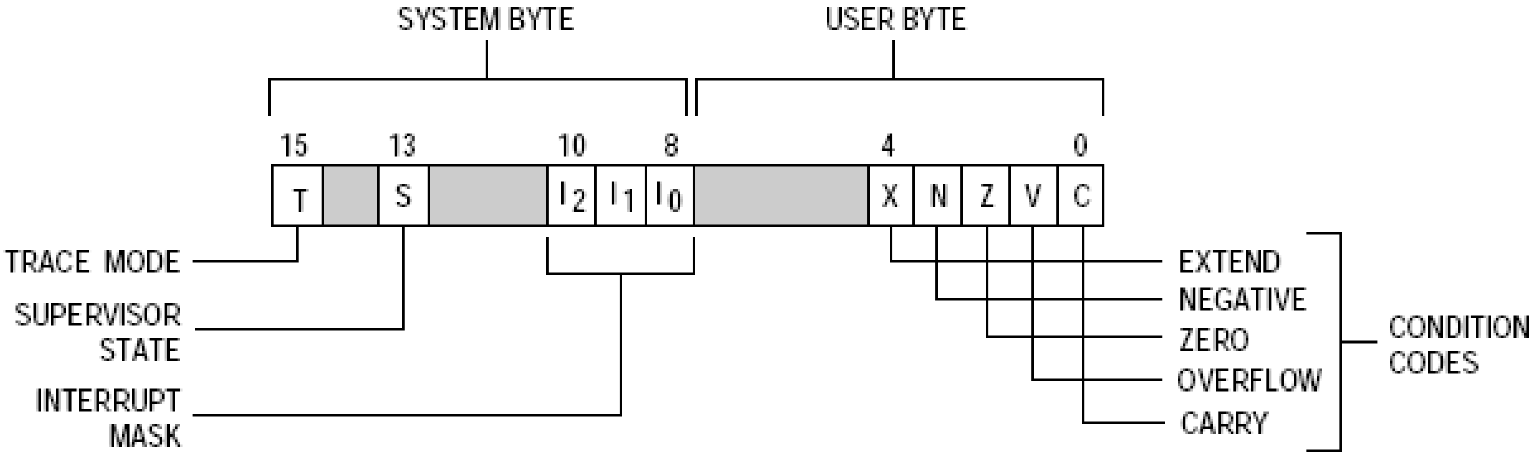
- Program Counter
- Instruction Decoding Register
- Memory Address Register
- Memory Buffer Register
- Program Status Word (PSW)

Program Status Word

- A set of bits
- Includes Condition Codes (CCR)
  - Sign of last result
  - Zero
  - Carry
  - Equal
  - Overflow
- Interrupt enable/disable, interrupt mask
- Supervisor

Program Status Word - Example

- Motorola 68000’s PSW



Supervisor Mode

- Intel ring zero
- Kernel mode
- Allows privileged instructions to execute
  - E.g., system call
- Used by operating system
- Not available to user programs

Condition Code Registers (CCR)

- Sets of individual bits
  - —Set by CPU as the result of operation
  - —e.g. result of last operation was zero, Z bit is to be ‘1’
- Can be read (implicitly) by programs
  - —e.g. Jump if zero
- Can not (usually) be set by programs
  - —Some instructions can set or clear the condition registers

Condition Code Registers (CCR)

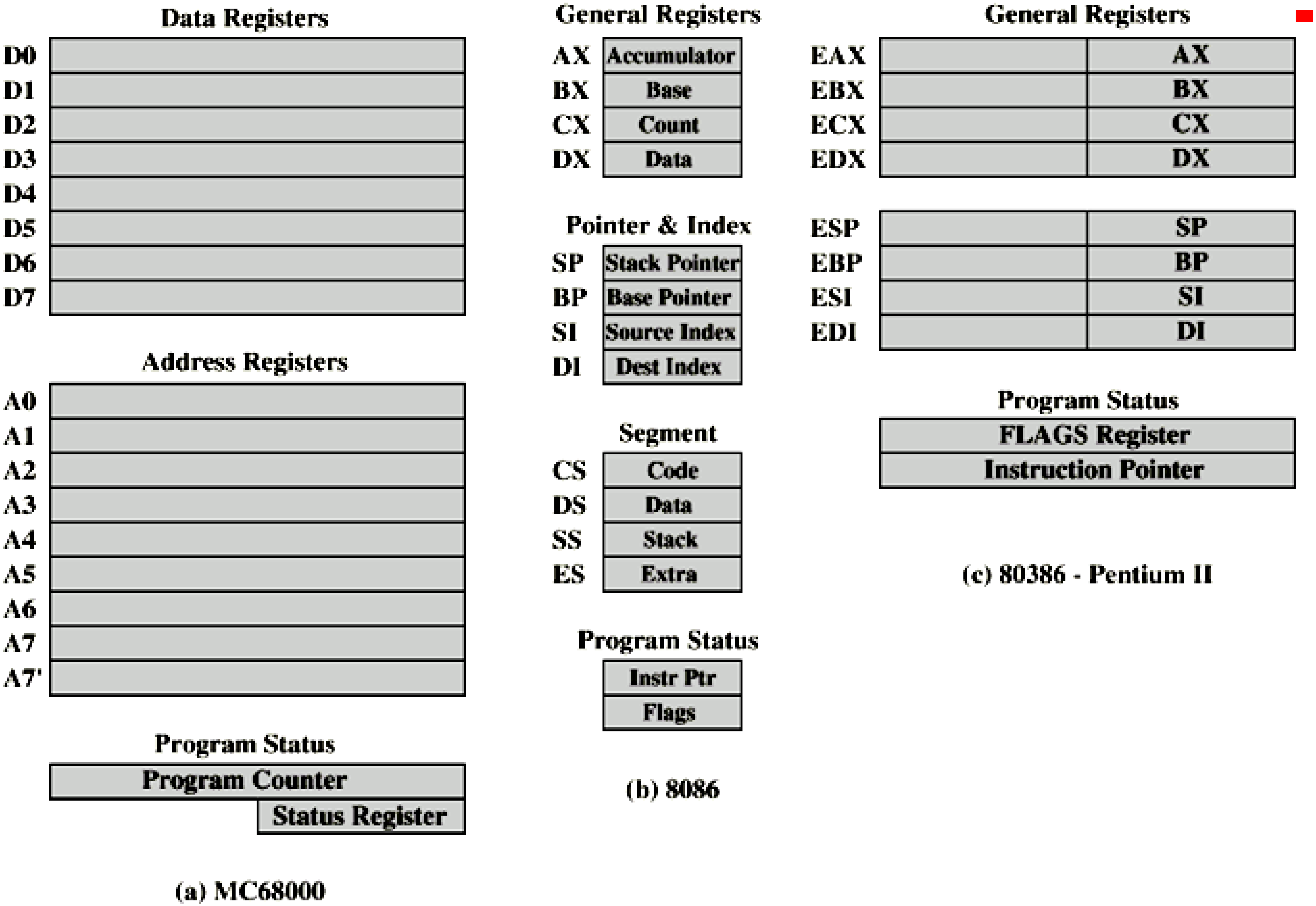
- Condition Code Register Bits N, Z, V, C
  - N (negative) bit is set if result of operation in negative (MSB = 1)
  - Z (zero) bit is set if result of operation is zero (All bits = 0)
  - V (overflow) bit is set if operation produced an overflow
  - C (carry) bit is set if operation produced a carry (borrow on subtraction)
- in MC68HC11E9, CCR is...
  - 8-bit register contains five condition code indicators (C,V,Z,N and H), two interrupt masking bits, (IRQ and XIRQ) and a stop disable bit (S)
  - automatically updated by most instructions.

H: Half carry flag

Other Registers

- May have registers pointing to:
  - Process control blocks
  - Interrupt vectors
  - Page table pointer
- CPU design and operating system design are closely linked

Example Register Organizations



Data Registers

D0
D1
D2
D3
D4
D5
D6
D7

Address Registers

A0
A1
A2
A3
A4
A5
A6
A7
A7'

Program Status

Program Counter
Status Register

(a) MC68000

General Registers

AX Accumulator
BX Base
CX Count
DX Data

Pointer & Index

SP Stack Pointer
BP Base Pointer
SI Source Index
DI Dest Index

Segment

CS Code
DS Data
SS Stack
ES Extra

Program Status

Instr Ptr
Flags

(b) 8086

General Registers

EAX AX
EBX BX
ECX CX
EDX DX

ESP SP
EBP BP
ESI SI
EDI DI

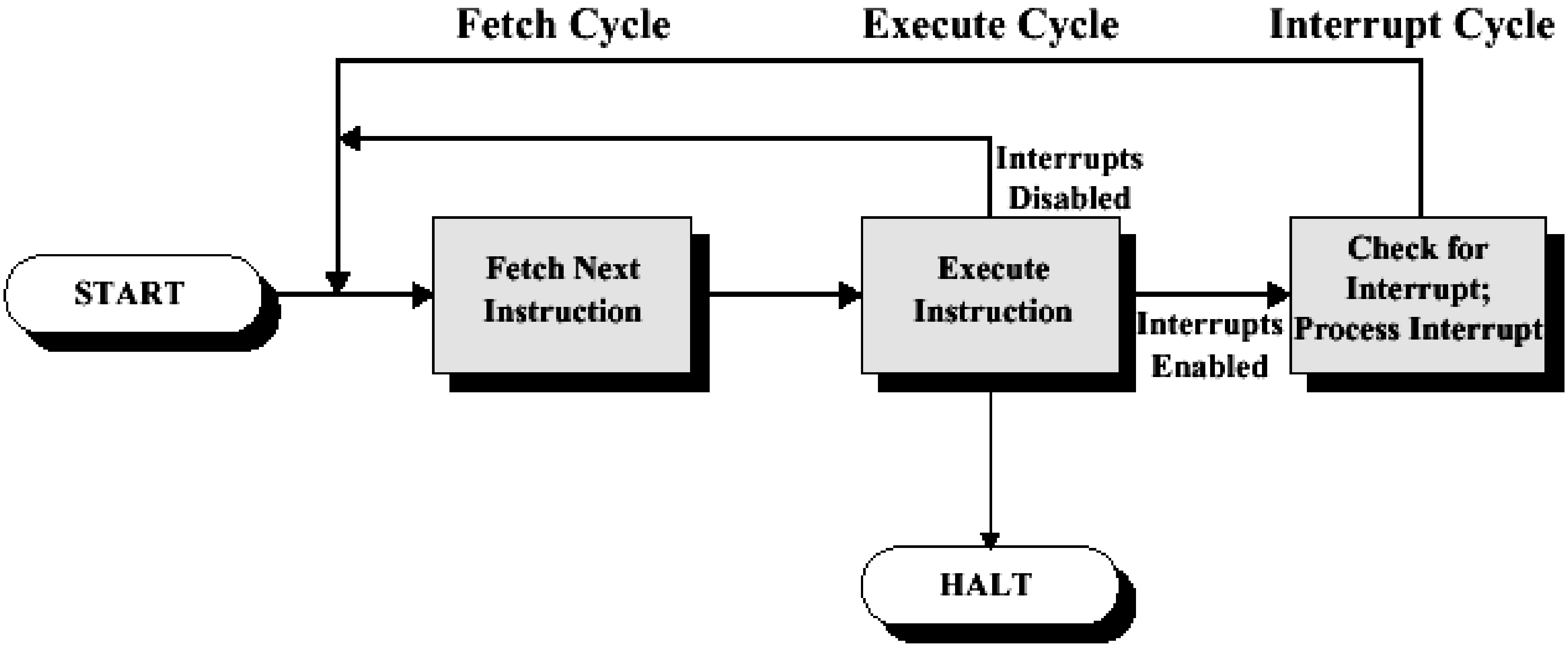
Program Status

FLAGS Register
Instruction Pointer

(c) 80386 - Pentium II

Instruction Cycle

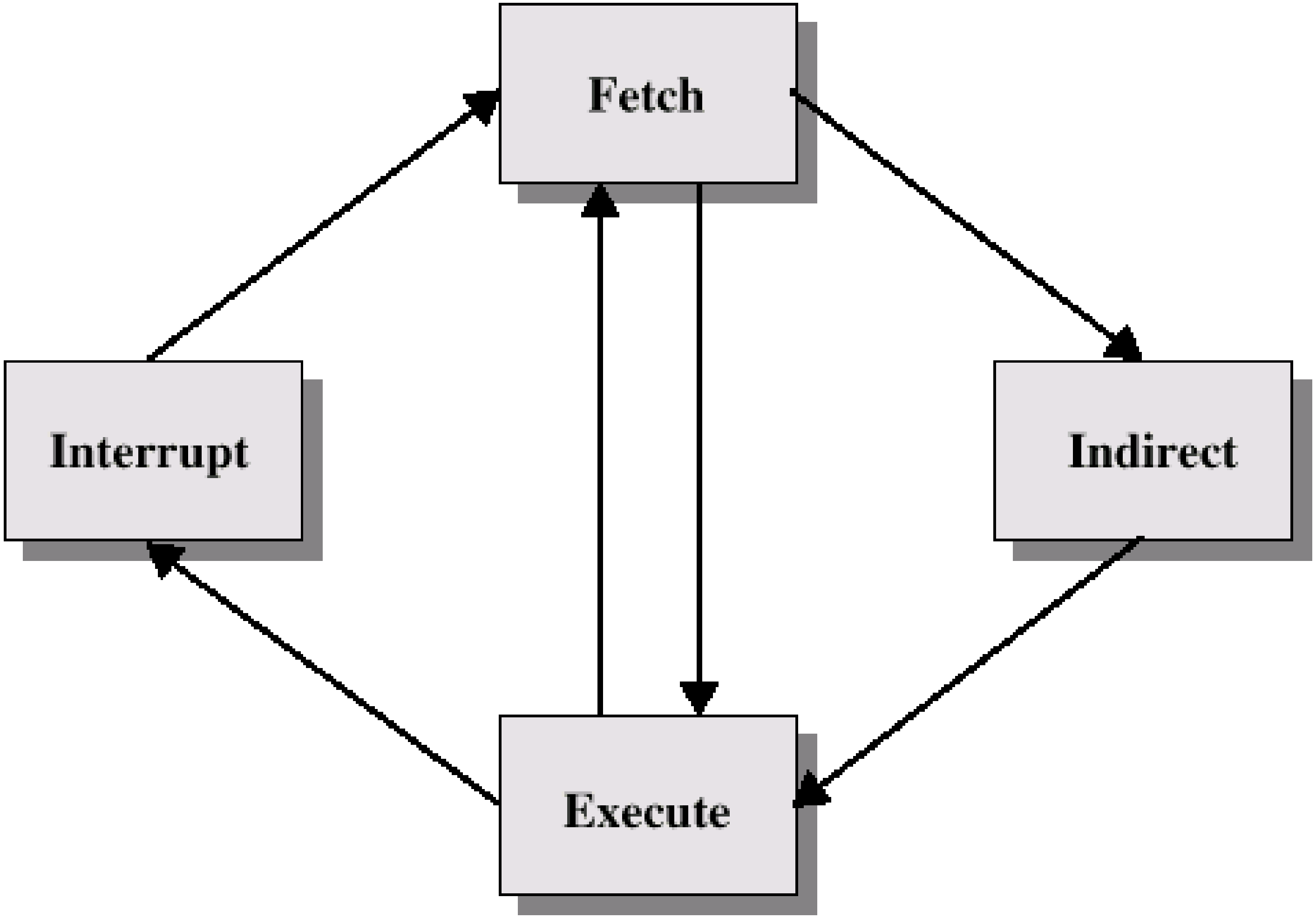
- Review



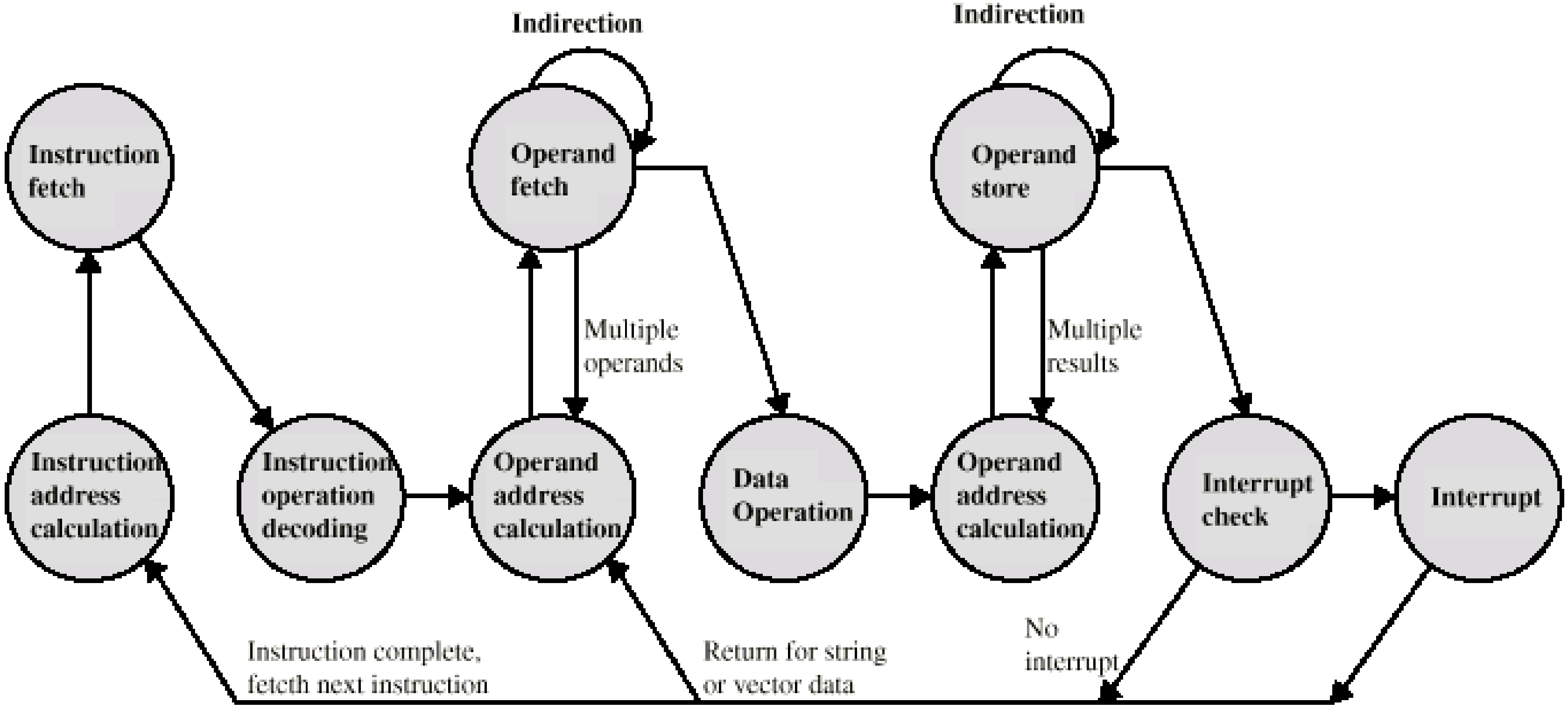
Indirect Cycle

- May require memory access to fetch operands
- Indirect addressing requires more memory accesses
- Can be thought of as additional instruction subcycle

Instruction Cycle with Indirect



Instruction Cycle State Diagram

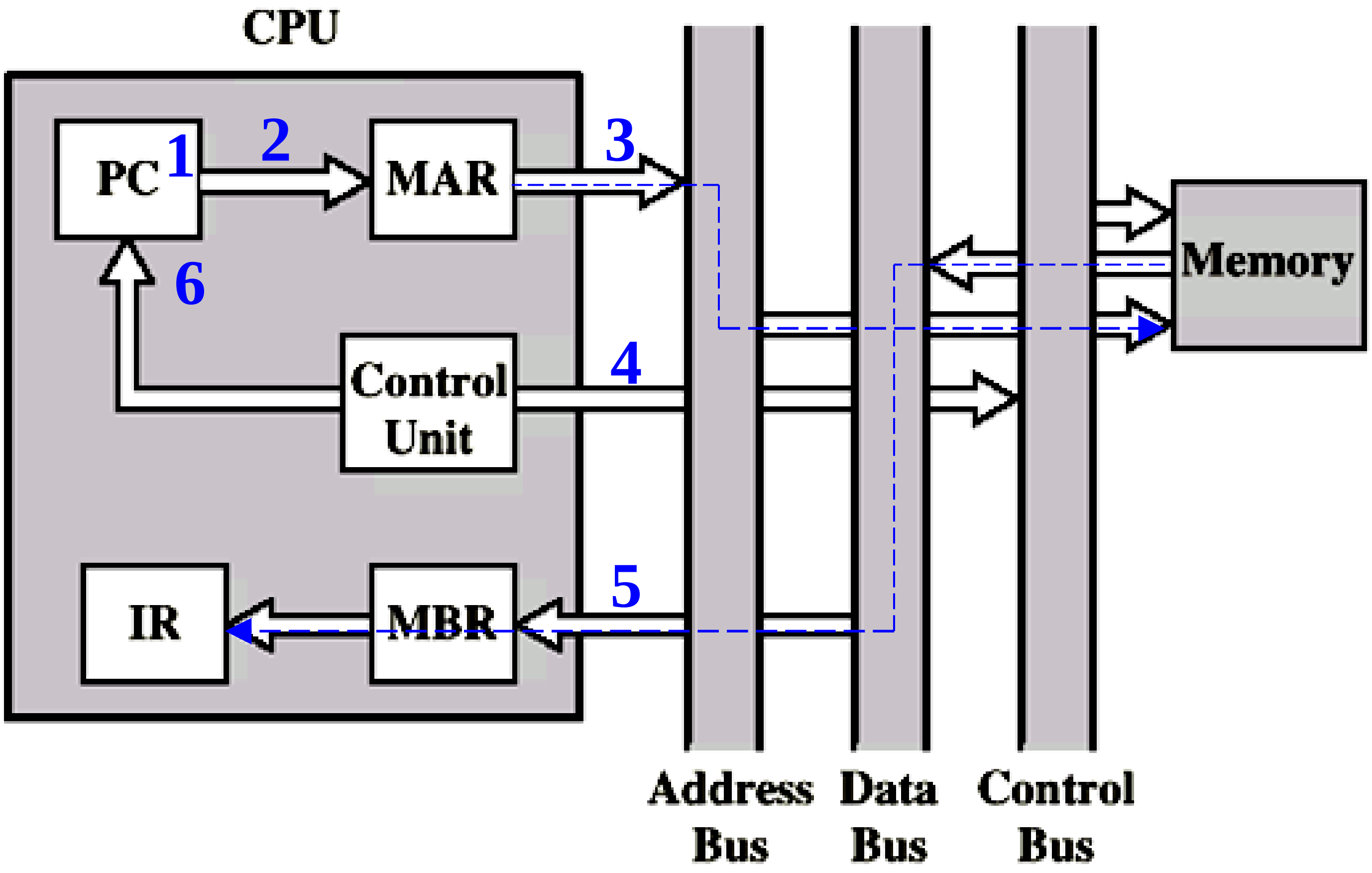

Indirection
Indirection
Instruction fetch
Operand fetch
Operand store
Multiple operands
Multiple results
Instruction address calculation
Instruction operation decoding
Operand address calculation
Data Operation
Operand address calculation
Interrupt check
Interrupt
Instruction complete, fetcth next instruction
Return for string or vector data
No interrupt

Data Flow (Instruction Fetch)

- Depends on CPU design
- In general:

- Fetch
  1. PC contains address of next instruction
  2. Address moved to MAR
  3. Address placed on address bus
  4. Control unit requests memory read
  5. Result placed on data bus, copied to MBR, then to IR
  6. Meanwhile PC incremented by 1

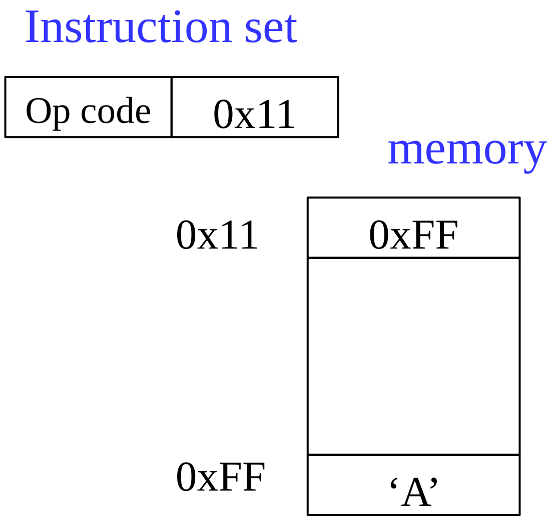
Data Flow (Fetch Diagram)



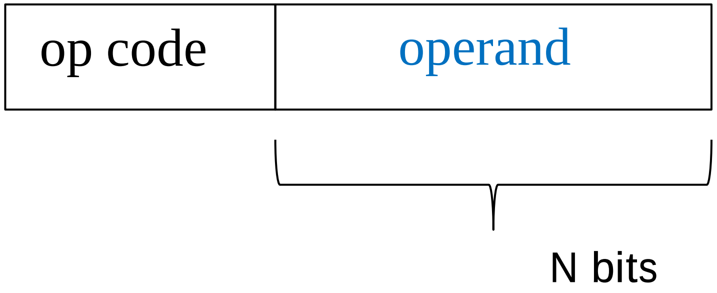
MBR = Memory buffer register
MAR = Memory address register
IR = Instruction register
PC = Program counter

Data Flow (Data Fetch)

- IR is examined
- If indirect addressing, indirect cycle is performed
    1. Right most N bits of MBR transferred to MAR
    2. Control unit requests memory read
    3. Result (address of operand) moved to MBR

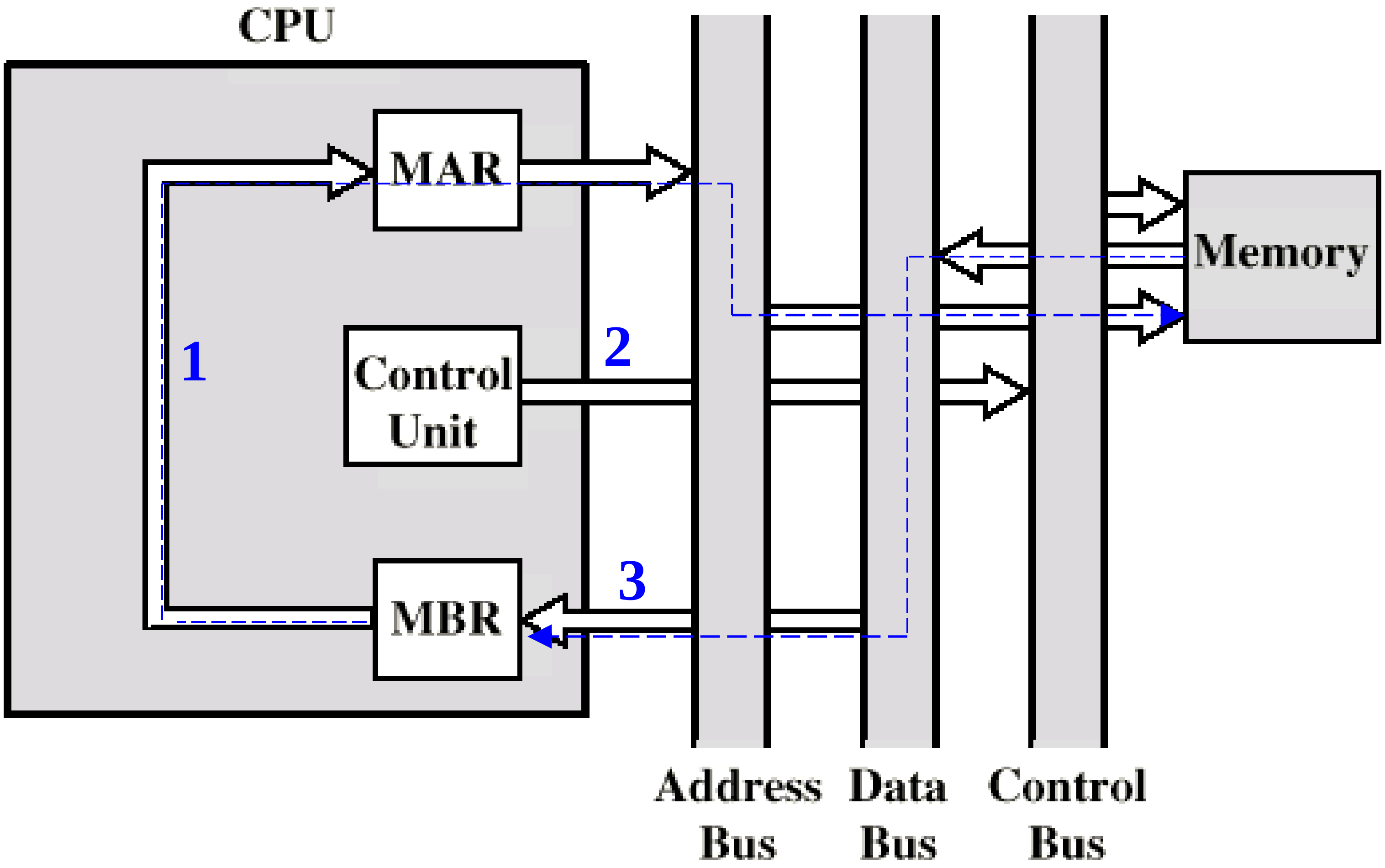


Instruction set format



4. The address of operand moved to MAR again!
5. Control unit request memory read
6. Result (data operand) moved to MBR

Data Flow (Indirect Diagram)



Data Flow (Execute)

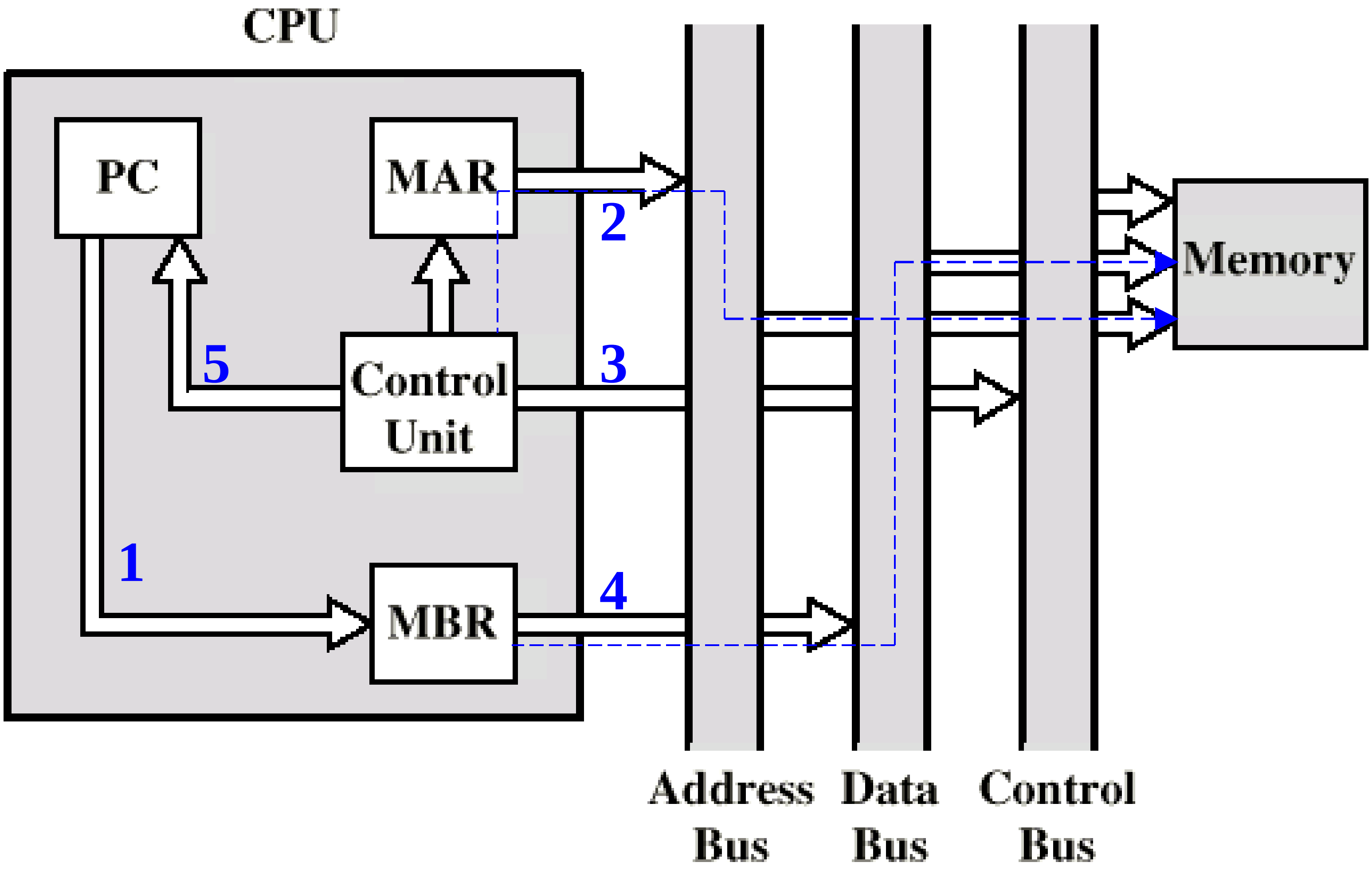
- May take many forms
- Depends on instruction being executed
- May include
  - Memory read/write
  - Input/Output
  - Register transfers
  - ALU operations

Data Flow (Interrupt)

- Simple, predictable
- Current PC saved to allow resumption after interrupt
- Include
  1. Contents of PC copied to MBR
  2. Special memory location (e.g. stack pointer) loaded to MAR
  3. Control unit: WRITE
  4. MBR written to memory
  5. PC loaded with address of interrupt handling routine

     (Next instruction (first of interrupt handler) can then be fetched)

Data Flow (Interrupt Diagram)



Prefetch

- Fetch accessing main memory
- Execution usually does not access main memory
- Can fetch next instruction during execution of current instruction
- Called instruction prefetch

Improved Performance

- But not doubled:
  - Fetch usually shorter than execution
    - Prefetch more than one instruction?
  - Any jump or branch means that prefetched instructions are not the required instructions
- Add more stages to improve performance

Pipelining

- Fetch instruction
- Decode instruction
- Calculate operands (i.e. EAs)
- Fetch operands
- Execute instructions
- Write result

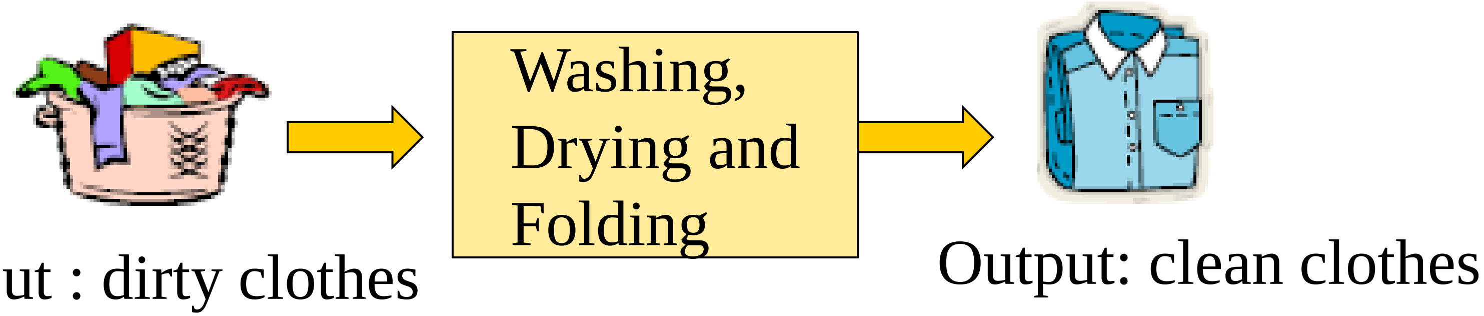
- Overlap these operations

What is Pipelining ?

- is a technique to increase their throughput (the number of instructions that can be executed in a unit of time)

The Concept of Pipelining

- The case of washing clothes

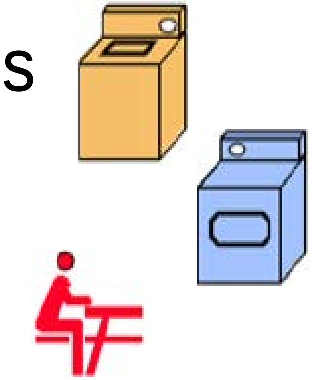


Washing, Drying and Folding

Input : dirty clothes

Output: clean clothes

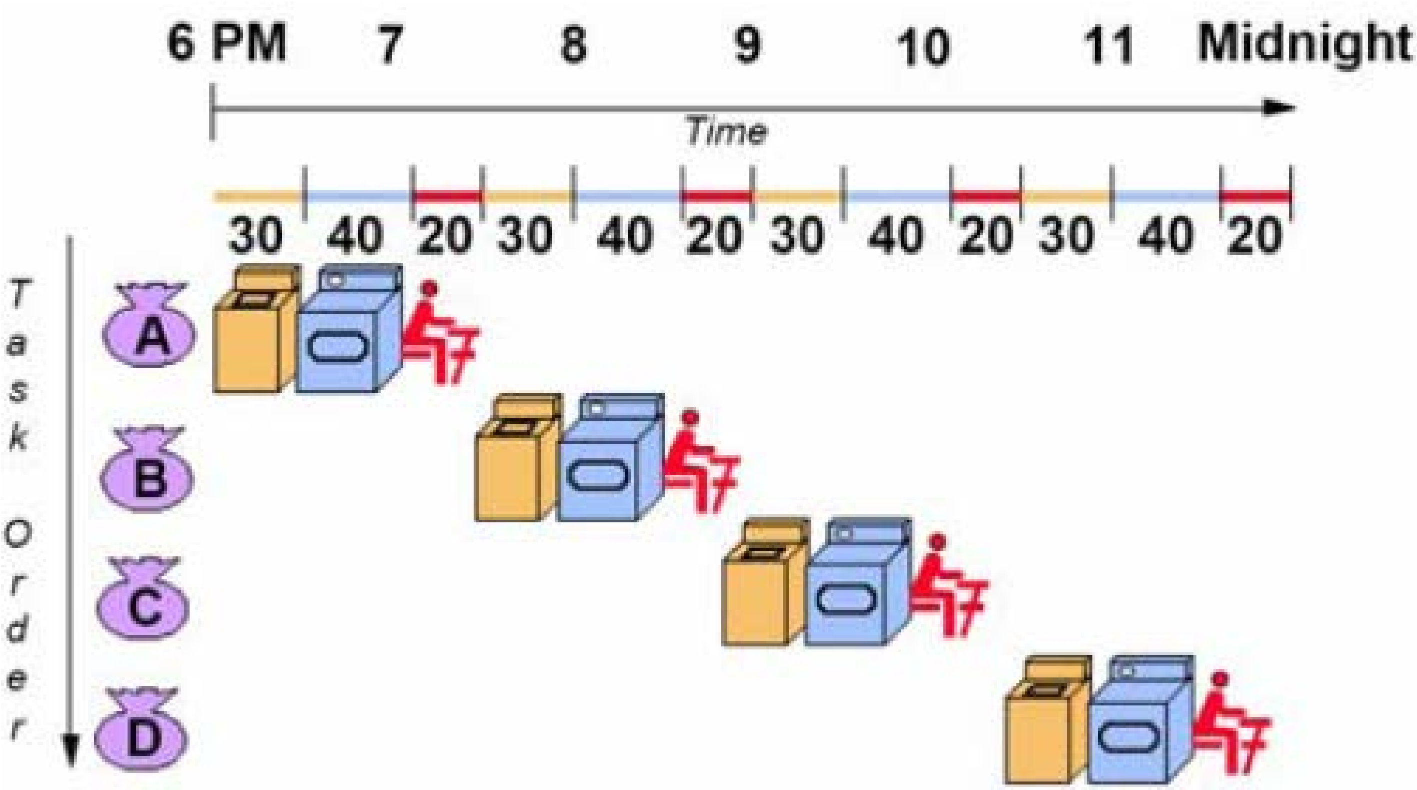
- Tasks
  - —Ann, Brian, Cathy and Dave each have one load of clothes to wash, dry and fold
- Washing takes 30 mins
- Drying takes 40 mins
- Folding takes 20 mins



Ref) http://www.seas.upenn.edu/~palsetia

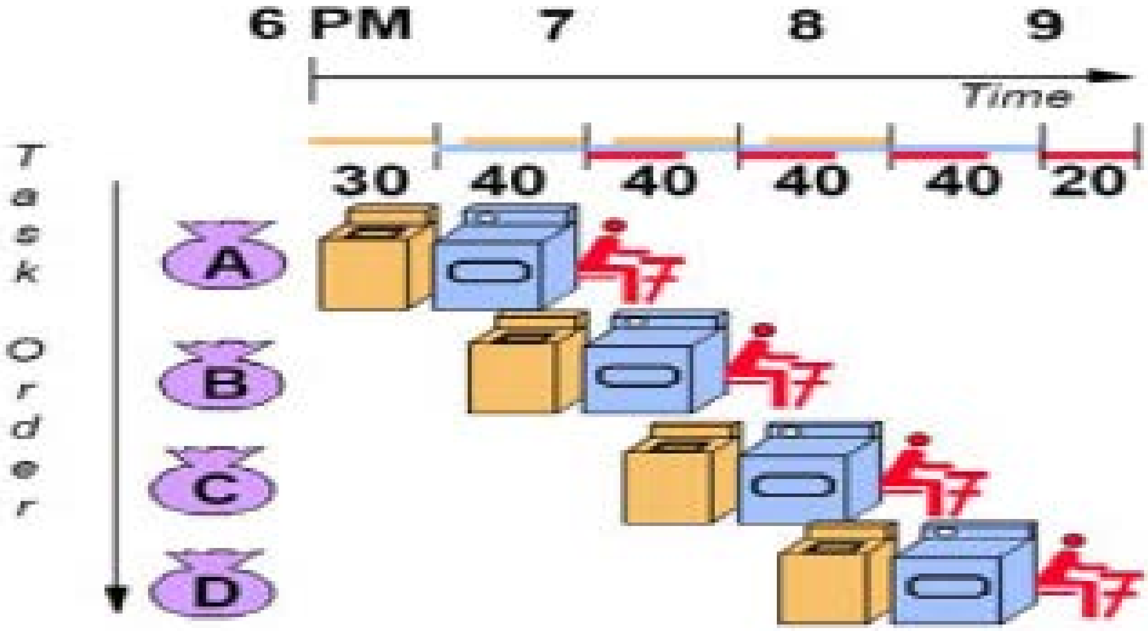
Case 1: Sequential Laundry

- Task order : Ann, Brian, Cathy and Dave



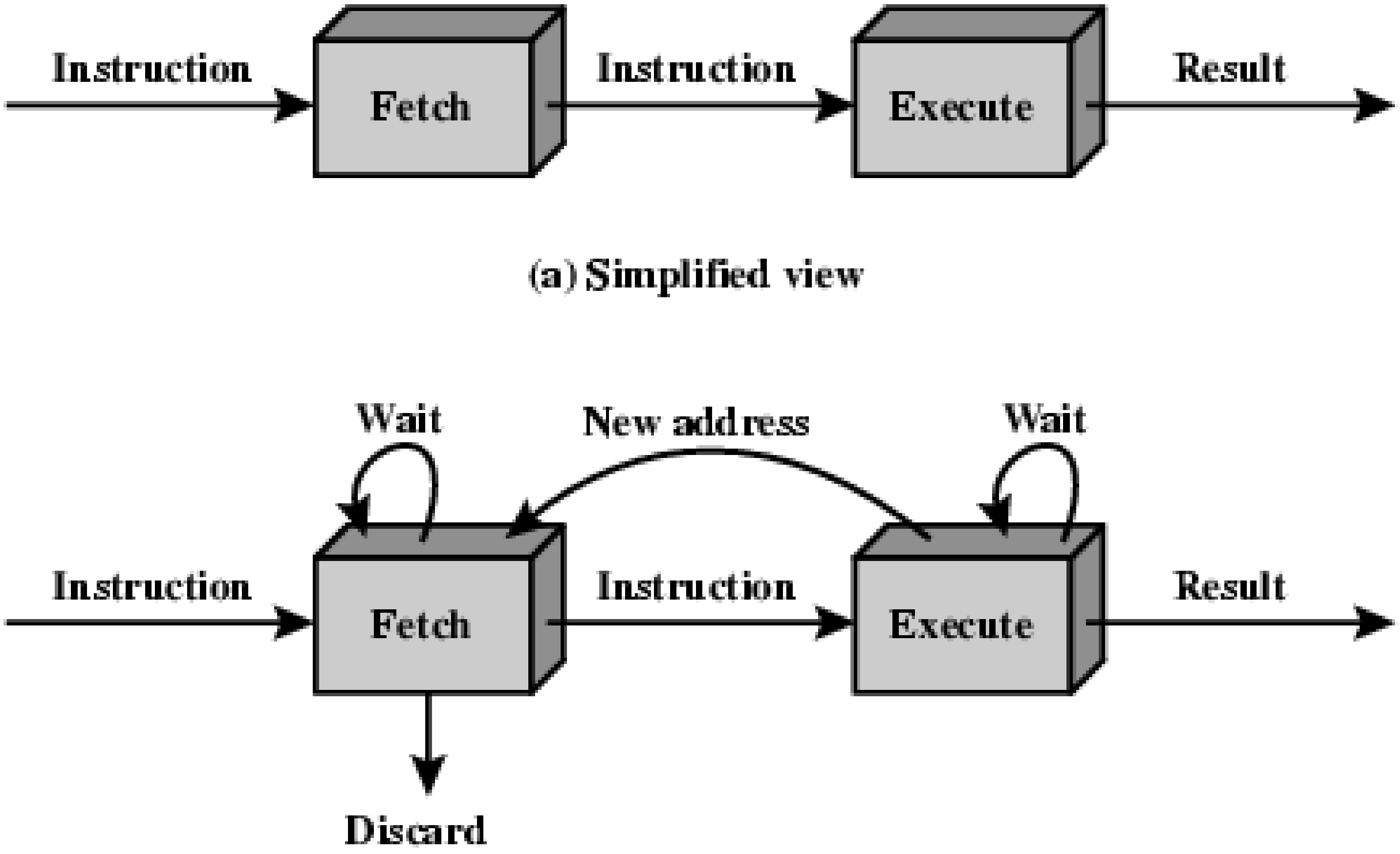
- Entire workload takes 6 hours to complete

Case 2: Pipelined Laundry



- Pipelined laundry takes only 3.5 hours
- Speedup = 6 hours / 3.5 hours = 1.7
- Pipelining did not reduce the completion time for one task but it helps the throughput of the entire workload

Two Stage Instruction Pipeline



(a) Simplified view

(b) Expanded view

Two Stage Instruction Pipeline (more)

- To fetch an instruction, accessing main memory is required
- Execution usually does not access main memory
- So it can fetch next instruction during execution of current instruction
- It is called instruction prefetch or fetch overlap
- Ideally instruction cycle time would be halved (if duration_F = duration_E ...)

Two Stage Instruction Pipeline (more)

- But in reality, it is not doubled because...
  - The execution time is longer than the fetch time
  - The conditional branch instruction makes the address of the next instruction to be fetched unknown

Two Stage Instruction Pipeline (more)

- Add more stages to improve performance
- Reduce time loss due to branching by guessing
  - If not branched

```
        use the prefetched instruction
    else
        discard the prefetched instruction
        fetch new instruction
```

The Six Stage Pipelining

- More stages ⇒ more speedup
  - FI: Fetch instruction
  - DI: Decode instruction
  - CO: Calculate operands (i.e. EAs)
  - FO: Fetch operands
  - EI: Execute instructions
  - WO: Write result
- Various stages are of nearly equal duration
- Overlap these operations

Timing Diagram for Instruction Pipeline Operation

Time →

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO	FO	EI	WO		
Instruction 8								FI	DI	CO	FO	EI	WO	
Instruction 9									FI	DI	CO	FO	EI	WO

τ= pipeline cycle time

k = number of stages : 6

n = number of instructions : 9

- To execute 9 instructions in 6 stage-pipeline device, it takes 14 time units to complete
- cf. in the case of no pipeline: 54 time units are required

Speedup of Pipelining

- Parameters

  τ = pipeline cycle time

  = time to advance a set of instructions one stage

  k = number of stages

  n = number of instructions

- Time to execute n instructions (In the case of no branch)

  $$T_k = [k + (n - 1)]\tau$$

- Time to execute *n* instructions without pipelining

  $$T_1 = nk\tau$$

Speedup of Pipelining

- Speedup of k-stage pipelining compared to w/o pipelining

$$S_k = \frac{Performance_{pipelining}}{Performance_{no\_pipelining}}$$

$$= \frac{T_1}{T_k} = \frac{nk\tau}{[k+(n-1)]\tau} = \frac{nk}{k+(n-1)}$$

$$n \to \infty, \ then\ k\ times\ speedup!$$

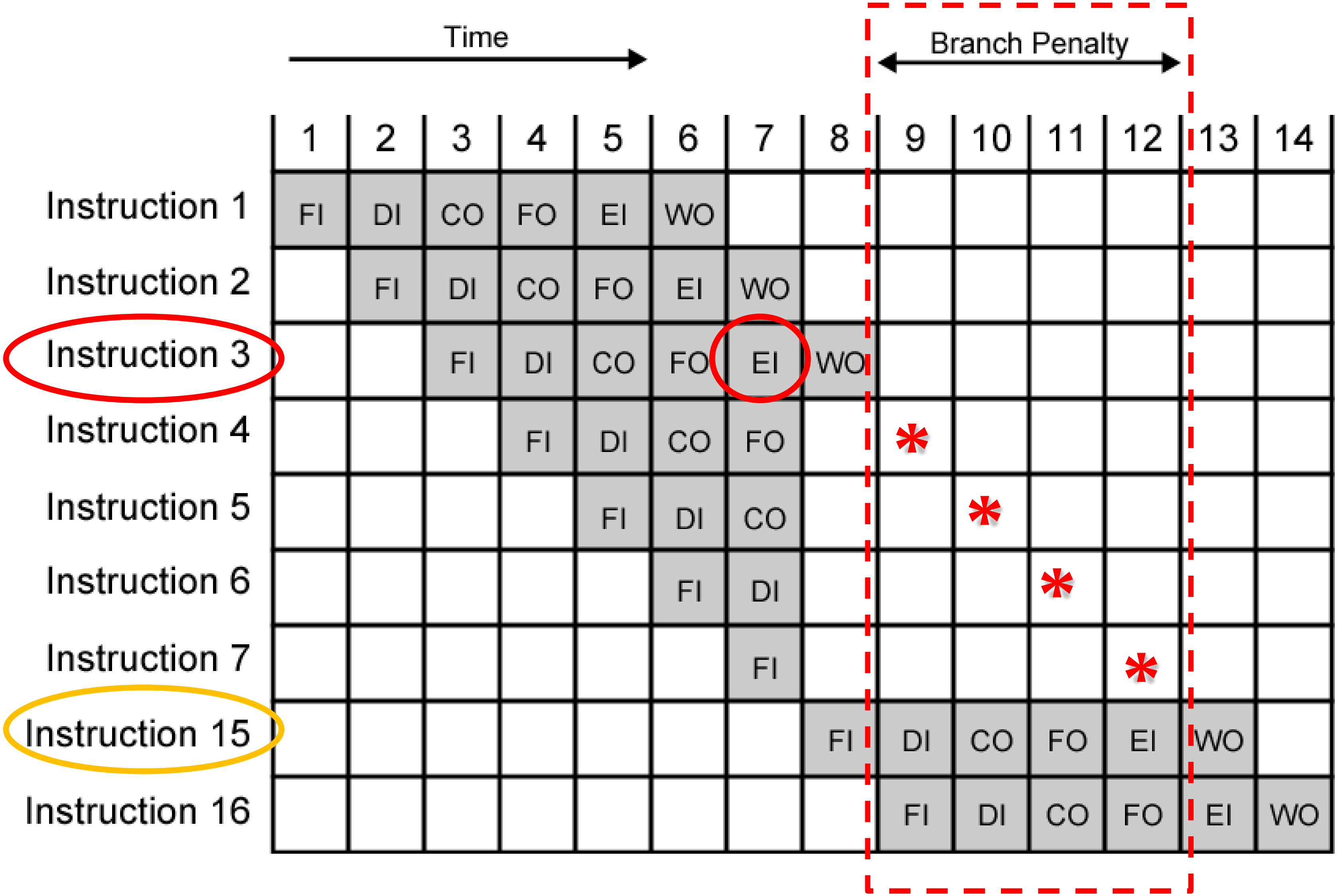
Limitation by Branching

- Conditional branch instructions can invalidate several instruction prefetches
- For example,
  - —Instruction 3 is a conditional branch to instruction 15
  - —Next instruction’s address won’t be known till instruction 3 is executed (at time unit 7)

    → pipeline must be cleared
  - —No instruction is finished from time units 9 to 12

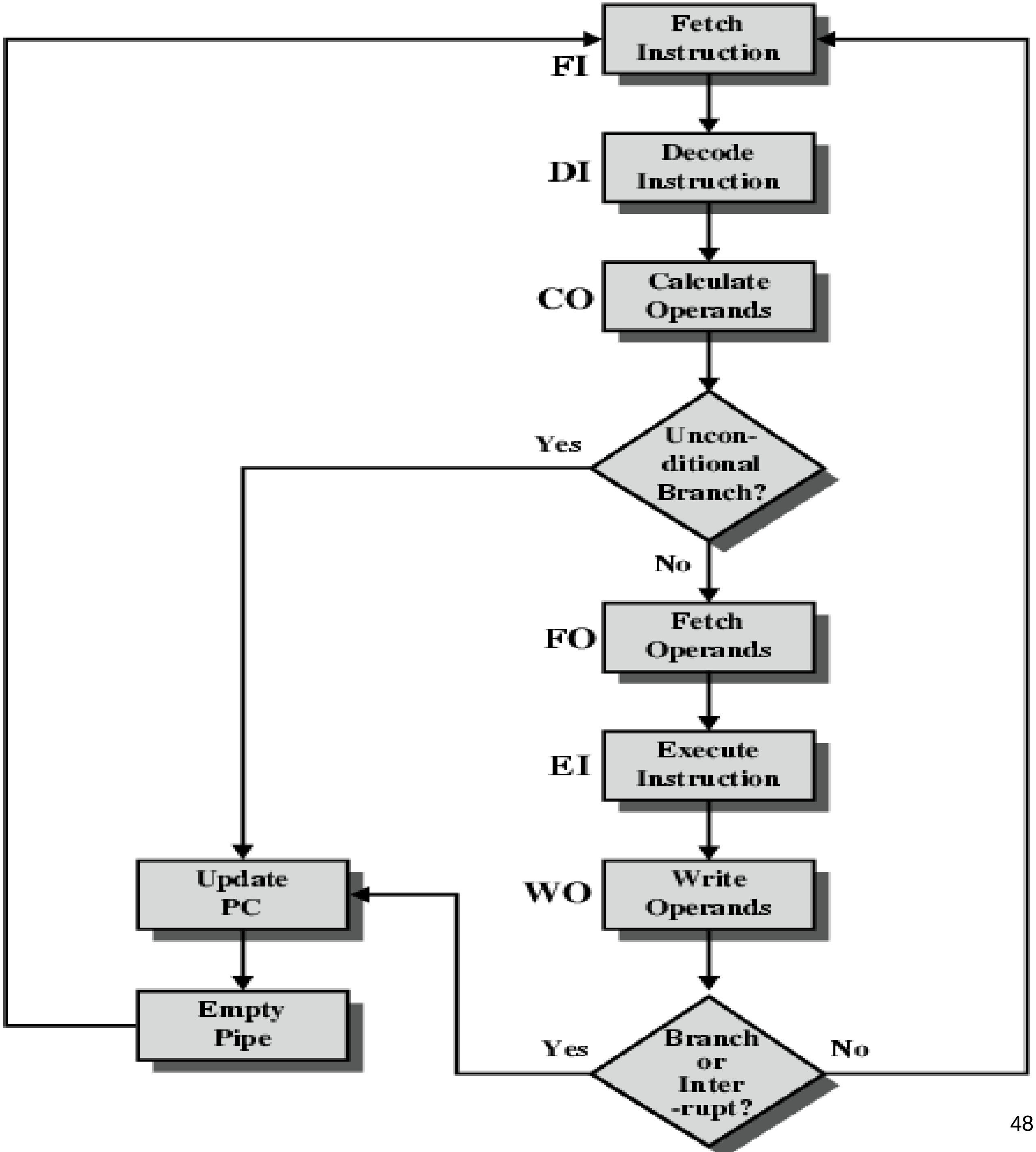
    → performance penalty

The Effect of a Conditional Branch on Instruction Pipeline Operation



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO		*					
Instruction 5					FI	DI	CO			*				
Instruction 6						FI	DI				*			
Instruction 7							FI					*		
Instruction 15								FI	DI	CO	FO	EI	WO	
Instruction 16									FI	DI	CO	FO	EI	WO

Six Stage Instruction Pipeline



Alternative Pipeline Depiction

Time

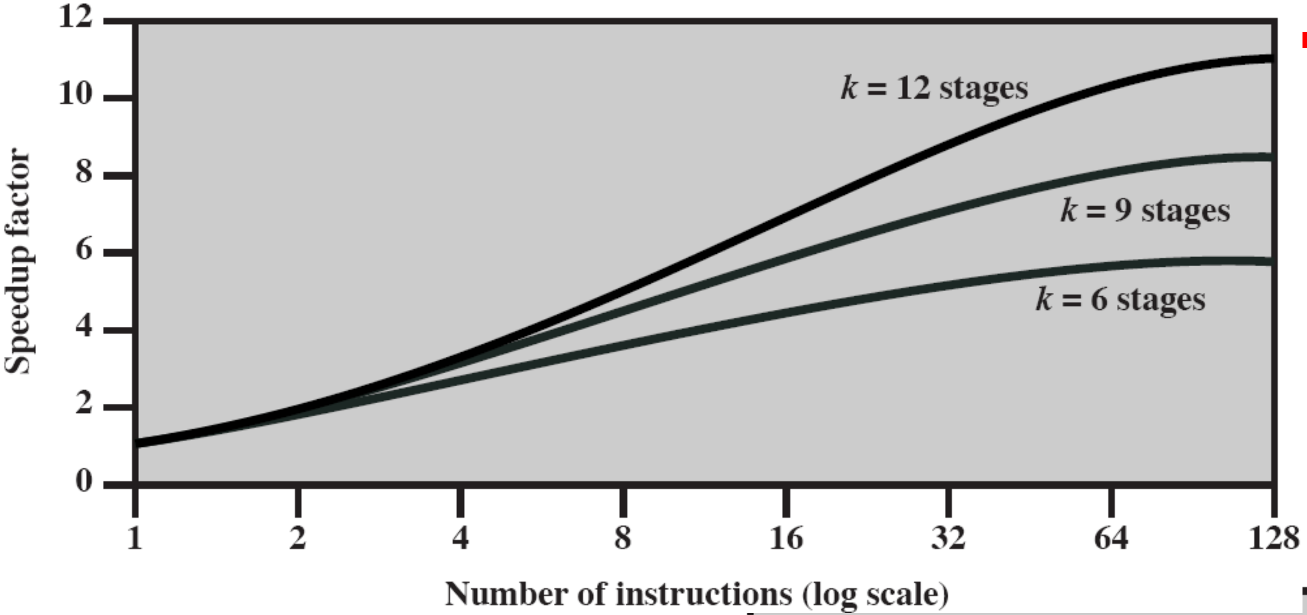
	FI	DI	CO	FO	EI	WO
1	I1					
2	I2	I1				
3	I3	I2	I1			
4	I4	I3	I2	I1		
5	I5	I4	I3	I2	I1	
6	I6	I5	I4	I3	I2	I1
7	I7	I6	I5	I4	I3	I2
8	I8	I7	I6	I5	I4	I3
9	I9	I8	I7	I6	I5	I4
10		I9	I8	I7	I6	I5
11			I9	I8	I7	I6
12				I9	I8	I7
13					I9	I8
14						I9

(a) No branches

	FI	DI	CO	FO	EI	WO
1	I1					
2	I2	I1				
3	I3	I2	I1			
4	I4	I3	I2	I1		
5	I5	I4	I3	I2	I1	
6	I6	I5	I4	I3	I2	I1
7	I7	I6	I5	I4	I3	I2
8	I15					I3
9	I16	I15				
10		I16	I15			
11			I16	I15		
12				I16	I15	
13					I16	I15
14						I16

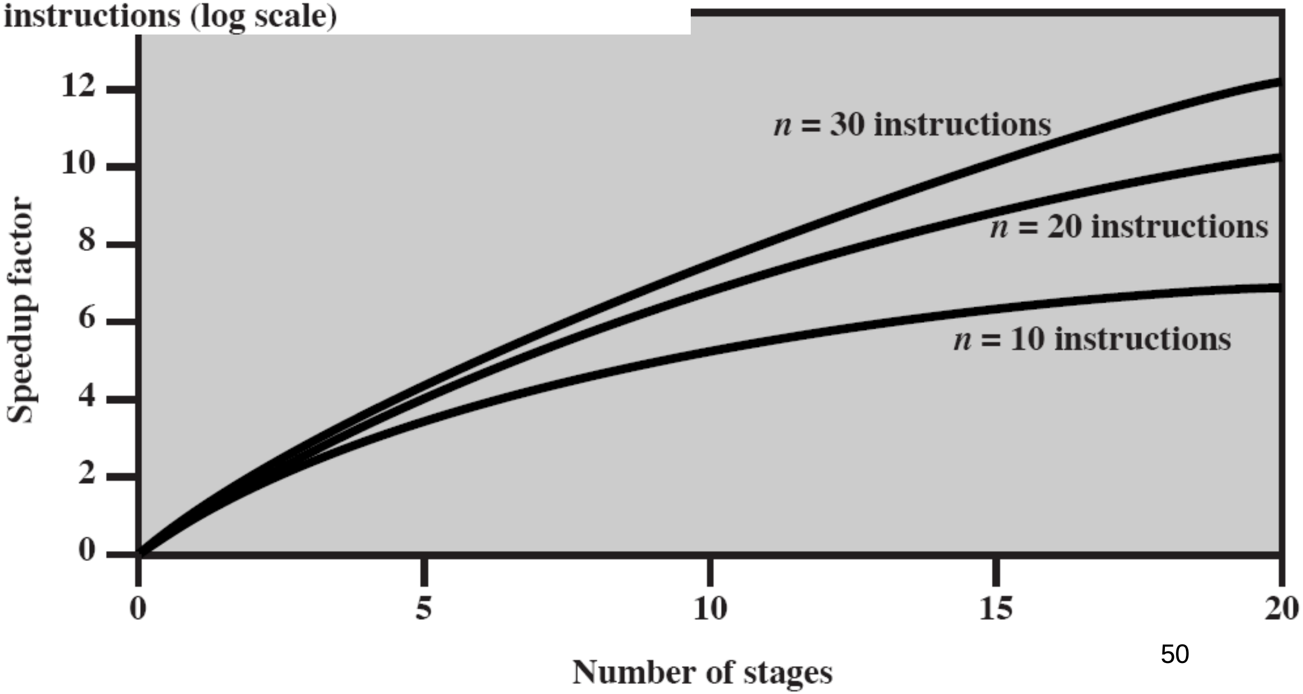
(b) With conditional branch

Speedup Factors with Instruction Pipelining



$n \to \infty,$

k *times speedup*!



Limitation by Data Dependencies

- Data needed by current instruction may depend on a previous instruction that is still in pipeline
- E.g., $A \leftarrow B + C$

  $D \leftarrow A + E$

Performance of Pipeline

- Ideally, more stages, more speedup
- However,
    - —more overhead in moving data between buffers
    - —more overhead in preparation
    - —more complex circuit for pipeline hardware

Dealing with Branches

- Multiple Streams
- Prefetch Branch Target
- Loop buffer
- Branch prediction
- Delayed branching

Multiple Streams

- Have two pipelines
- Prefetch each branch into a separate pipeline
- Use appropriate pipeline
- Problems
  - Leads to bus & register contention
  - Multiple branches lead to further pipelines being needed

다수의 pipeline 열에서 fetch를 하니까

If … then
A
else B

Prefetch Branch Target

- Target of branch is prefetched in addition to instructions following branch
- Keep target until branch is executed
- Used by IBM 360/91

Loop Buffer

- Very fast memory
- Maintained by fetch (IF) stage of pipeline
- Check buffer before fetching from memory
- Very good for small loops or jumps
- If a branch is to be taken
  - Hardware checks whether the target is in buffer

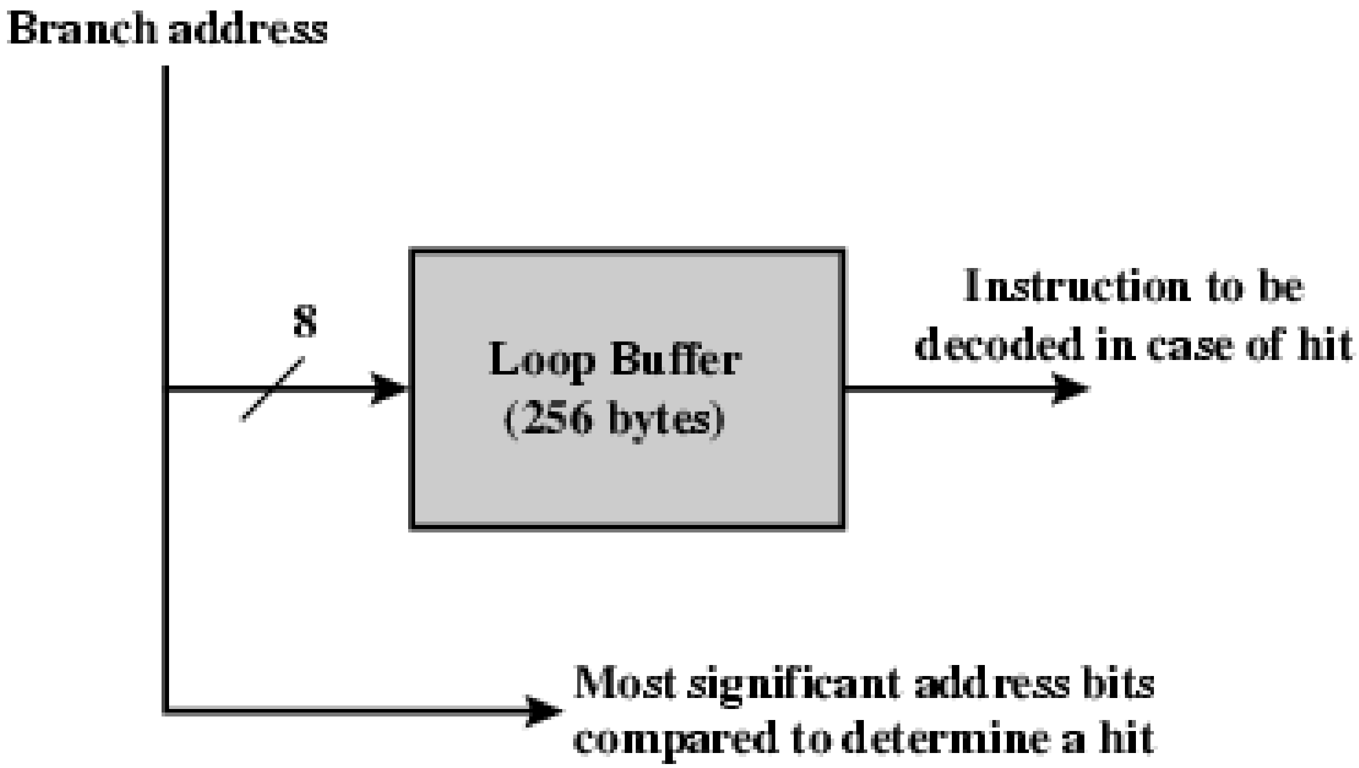
```
If YES then
  next instruction is fetched from the buffer
else
  fetch from memory
```

- If buffer is big enough to contain entire loop, instructions in the loop need to be fetched from memory only once at the first iteration
- c.f. cache (what is different ?)
- Used by CRAY-1

Loop Buffer – Benefits

- With the use of prefetching, the loop buffer will contain some instructions sequentially ahead of the current instruction fetch address
  - Thus, instructions fetched in sequence will be available without the usual memory access time
- If a branch occurs to a target just a few locations ahead of the address of the branch instruction, the target will already be in the buffer
  - This is useful for the rather common occurrence of IF-THEN and IF-THEN-ELSE sequence
- This strategy is well suited to dealing with loops, or iterations; hence the name loop buffer. If the loop buffer is large enough to contain all the instructions in a loop, then those instructions need to be fetched from memory only once, for the first iteration.
  - For subsequent iterations, all the needed instructions are already in the buffer.

Loop Buffer Diagram



Branch Prediction (1)

- Predict whether a branch will be taken
- If the prediction is right

  ⇒ No branch penalty
- If the prediction is wrong

  ⇒ Empty pipeline

  Fetch correct instruction

  ⇒ Branch penalty

Branch Prediction (2)

- Predict techniques
- Static
  - Predict never taken
  - Predict always taken
  - Predict by opcode
- Dynamic
  - Taken/not taken switch
  - Branch history table

Branch Prediction (3)

- Predict never taken
  - Assume that jump will not happen
  - Always fetch next instruction
  - 68020 & VAX 11/780
  - VAX will not prefetch after branch if a page fault would result (O/S v CPU design)
- Predict always taken
  - Assume that jump will happen
  - Always fetch target instruction

Branch Prediction (4)

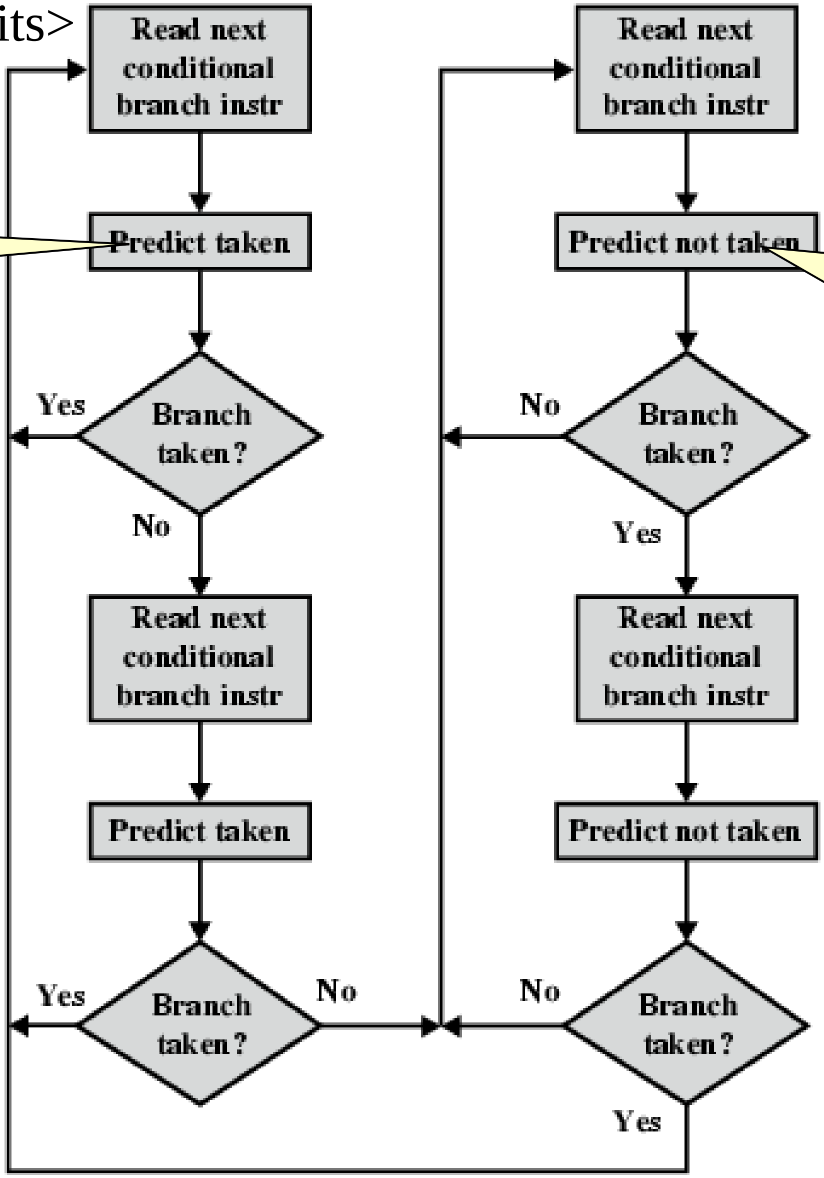
- Predict by Opcode
  - —Some instructions are more likely to result in a jump than others
  - —Can get up to 75% success
- Taken/Not taken switch
  - —Based on previous history
  - —Good for loops (why?)
- Branch history table
  - —Like a cache to look up

Branch Prediction Flowchart

< use of the two history bits>

Predict always taken:
Assume that jump will happen.
Always fetch target instruction.

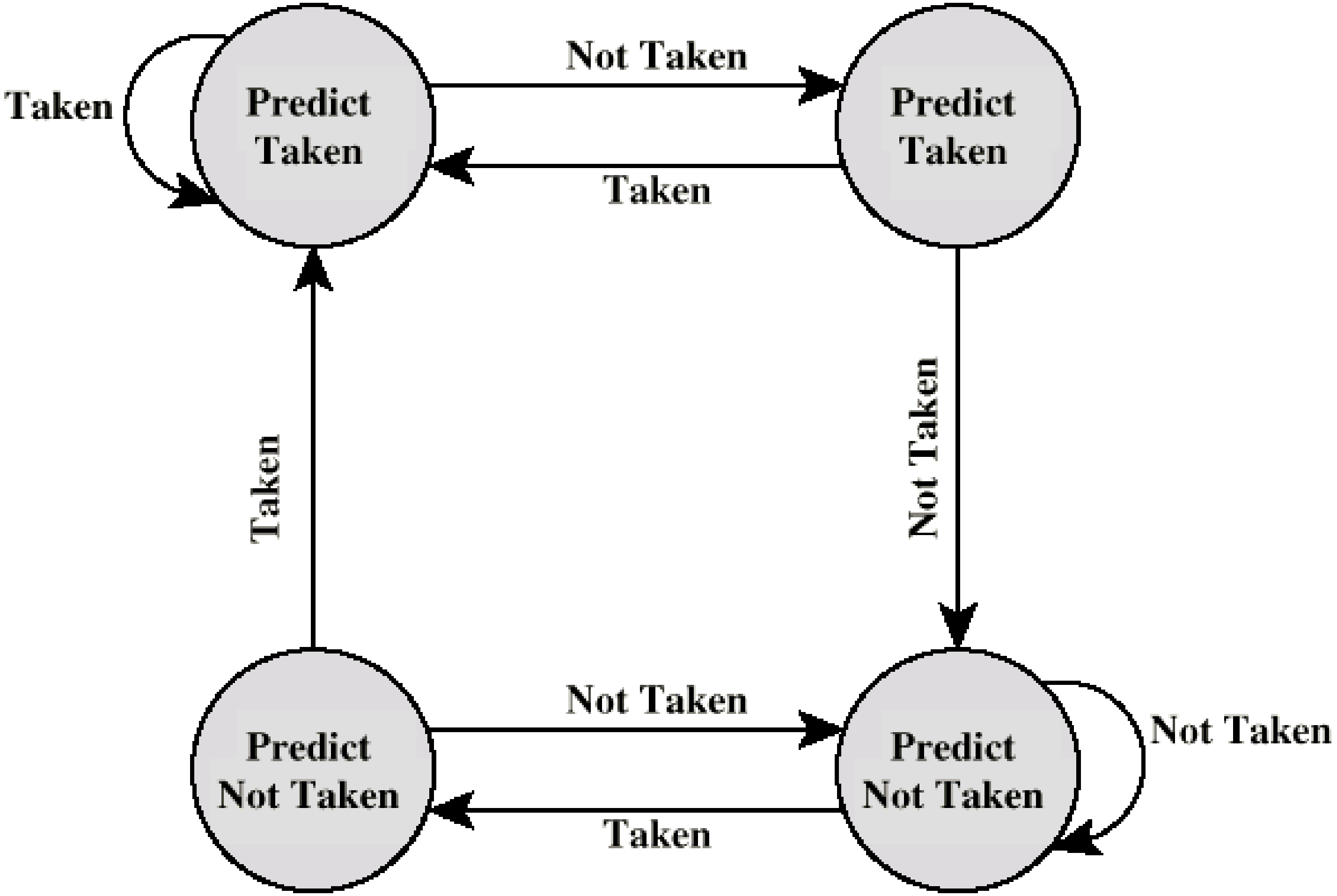
Predict never taken
: Assume that jump will not happen. Always fetch next instruction.


Read next conditional branch instr
Predict taken
Branch taken?
Yes
No
Read next conditional branch instr
Predict taken
Branch taken?
Yes
No
Read next conditional branch instr
Predict not taken
Branch taken?
No
Yes
Read next conditional branch instr
Predict not taken
Branch taken?
No
Yes

Two consecutive wrong predictions will change the prediction decision

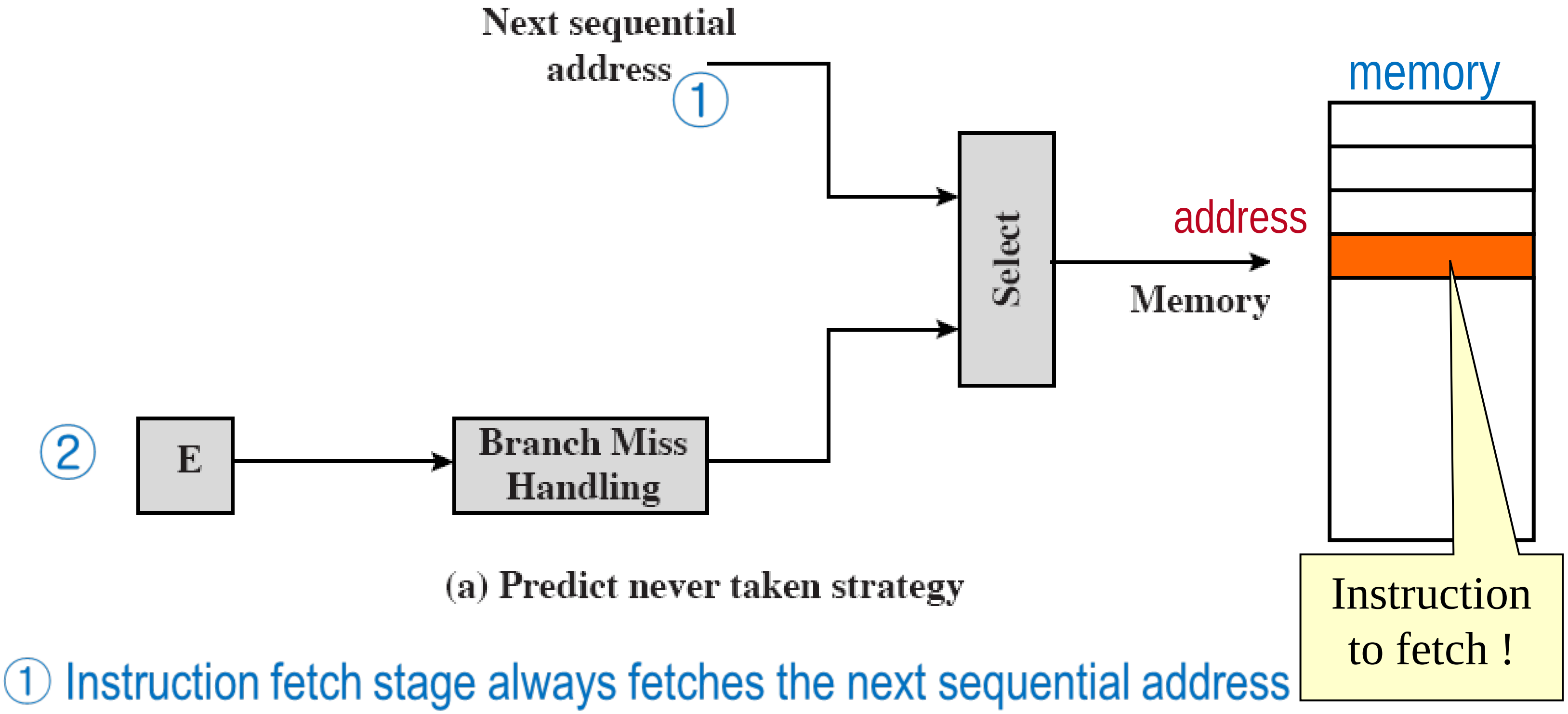
Finite State Machine for Branch Prediction

< use of the two history bits>



Dealing With Branches – Predict never taken

"predict never taken" 전략에선 branch가 항상 일어나지 않는다고 가정하여, next sequential address에 해당하는 memory 위치를 access함. → 이 예측이 틀린 경우(즉 branch가 일어난경우: E, 즉 실행 단계에서 이를 확인할 수 있음), target address에 해당하는 memory 내용을 읽을 것임

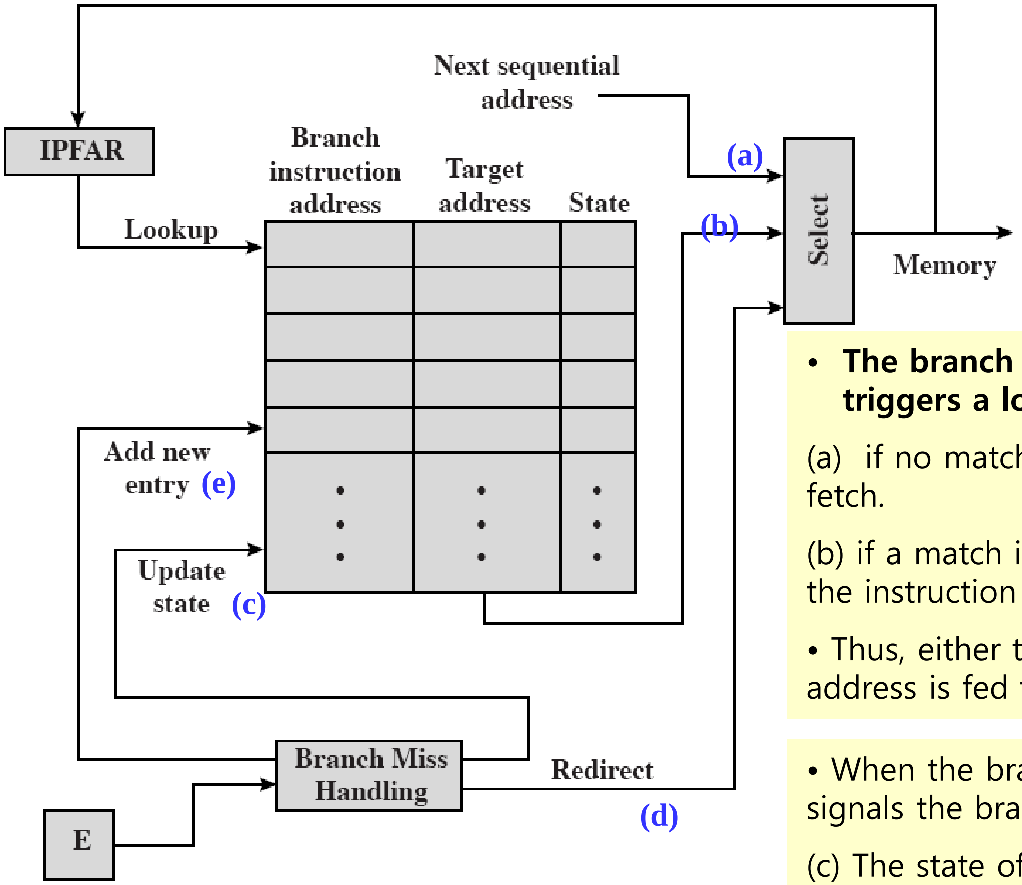


(a) Predict never taken strategy

-① Instruction fetch stage always fetches the next sequential address

-② if a branch is taken, the processor detects this & instructs that the next instruction be fetched from the target address (& flush the pipeline)

Dealing With Branches – Branch history table



(b) Branch history table strategy

IPFAR: Instruction PreFetch Address Register

- **The branch history table is treated as a cache. Each prefetch triggers a lookup in the branch history table**

(a) if no match is found, the next sequential address is used for the fetch.

(b) if a match is found, a prediction is made based on the state of the instruction

- Thus, either the next sequential address or the branch target address is fed to select logic.

- When the branch instruction is executed, the execute stage **(E)** signals the branch history table logic with the result.

(c) The state of the instruction is updated to reflect a correct or incorrect prediction.

(d) If the prediction is incorrect, the select logic is redirected to the correct address for the next fetch

(e) When a conditional branch instruction is encountered that is not in the table, it is added to the table and one of the existing entries is discarded, using one of the cache replacement algorithms discussed previously.

Delayed Branching

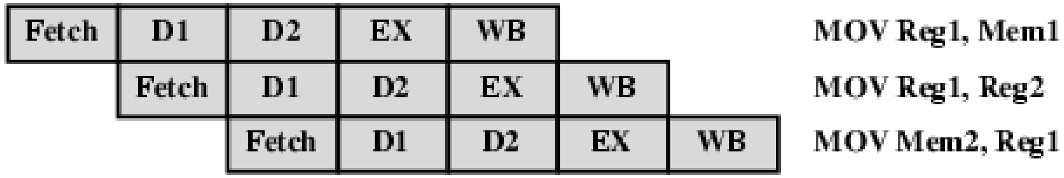
- Do not take jump until you have to
- Rearrange instructions so that branch instruction occurs later than actually desired (Chapter 13)

Q&A

Intel 80486 Pipelining

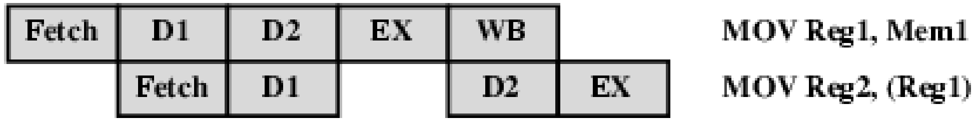
- Fetch
  - From cache or external memory
  - Put in one of two 16-byte prefetch buffers
  - Fill buffer with new data as soon as old data consumed
  - Average 5 instructions fetched per load
  - Independent of other stages to keep buffers full
- Decode stage 1
  - Opcode & address-mode info
  - At most first 3 bytes of instruction
  - Can direct D2 stage to get rest of instruction
- Decode stage 2
  - Expand opcode into control signals
  - Computation of complex address modes
- Execute
  - ALU operations, cache access, register update
- Writeback
  - Update registers & flags
  - Results sent to cache & bus interface write buffers

80486 Instruction Pipeline Examples



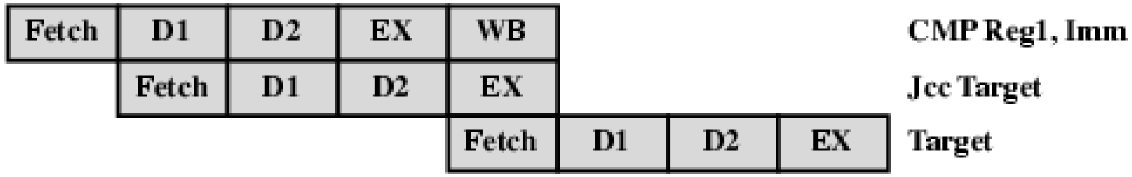
Fetch	D1	D2	EX	WB			MOV Reg1, Mem1
	Fetch	D1	D2	EX	WB		MOV Reg1, Reg2
		Fetch	D1	D2	EX	WB	MOV Mem2, Reg1

(a) No Data Load Delay in the Pipeline



Fetch	D1	D2	EX	WB		MOV Reg1, Mem1
	Fetch	D1		D2	EX	MOV Reg2, (Reg1)

(b) Pointer Load Delay

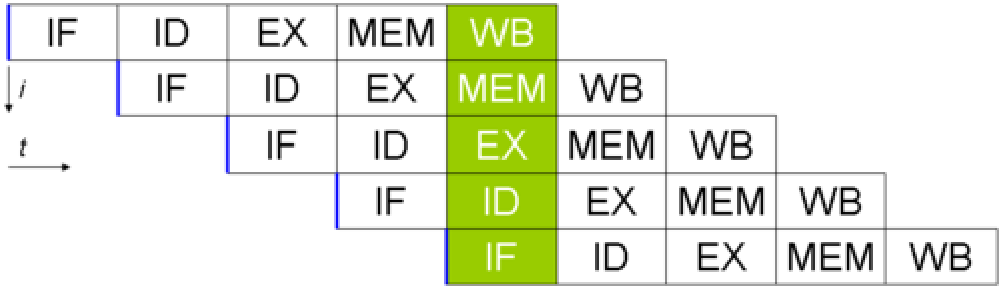


Fetch	D1	D2	EX	WB				CMP Reg1, Imm
	Fetch	D1	D2	EX				Jcc Target
				Fetch	D1	D2	EX	Target

(c) Branch Instruction Timing

Appendix: Pipelining Hardware ref) wikipedia

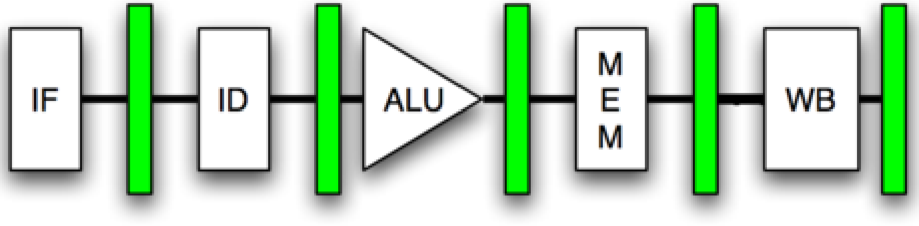
- Given our multicycle processor, what if we wanted to overlap our execution, so that up to 5 instructions could be processed at the same time? Let's contract our timing diagram a little bit to show this idea:



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

- As this diagram shows, each element in the processor is active in every cycle, and the instruction rate of the processor has been increased by 5 times! The question now is, what additional hardware do we need in order to perform this task? We need to add storage registers between each pipeline state to store the partial results between cycles, and we also need to reintroduce the redundant hardware from the single-cycle CPU. We can continue to use a single memory module (for instructions and data), so long as we restrict memory read operations to the first half of the cycle, and memory write operations to the second half of the cycle (or vice-versa). We can save time on the memory access by calculating the memory addresses in the previous stage.

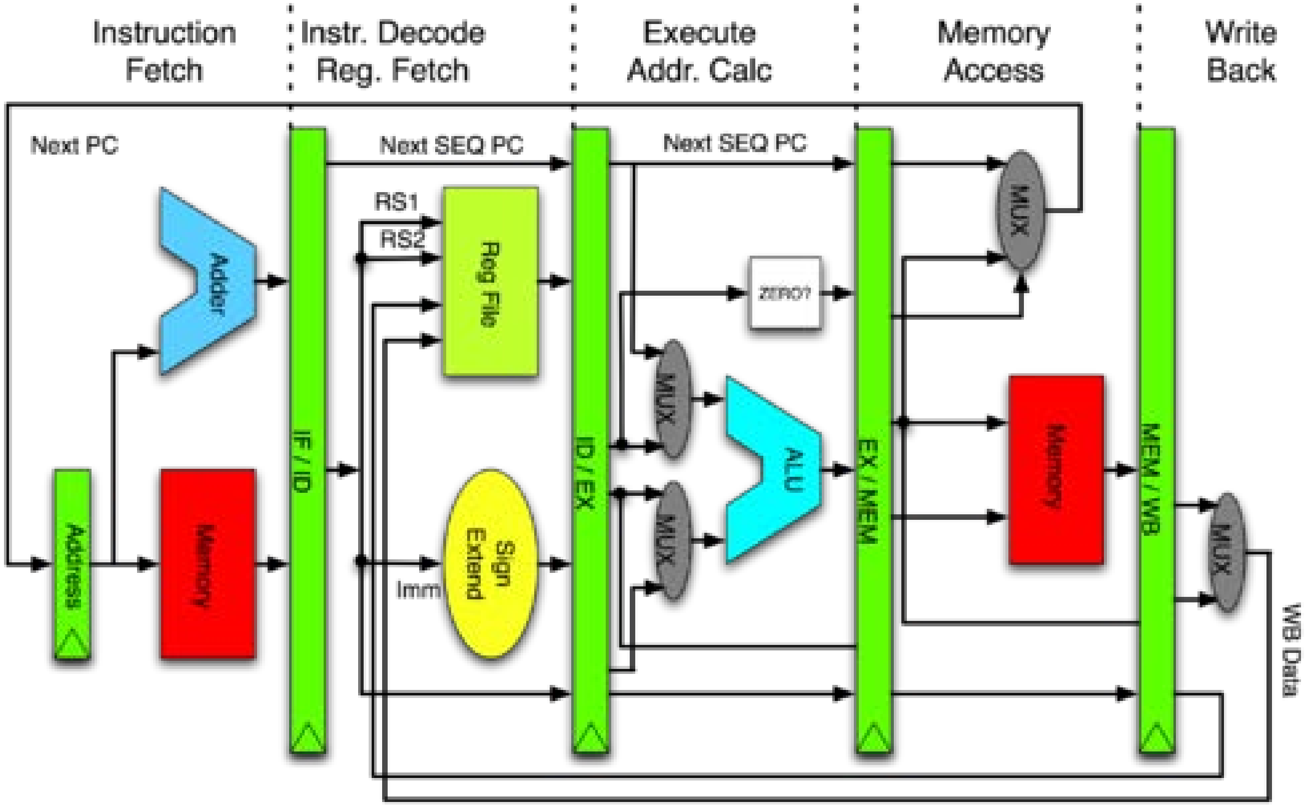
Appendix: Pipelining Hardware

- 

- The registers would need to hold the data from the pipeline at that point, and also the necessary control codes to operate the remainder of the pipeline

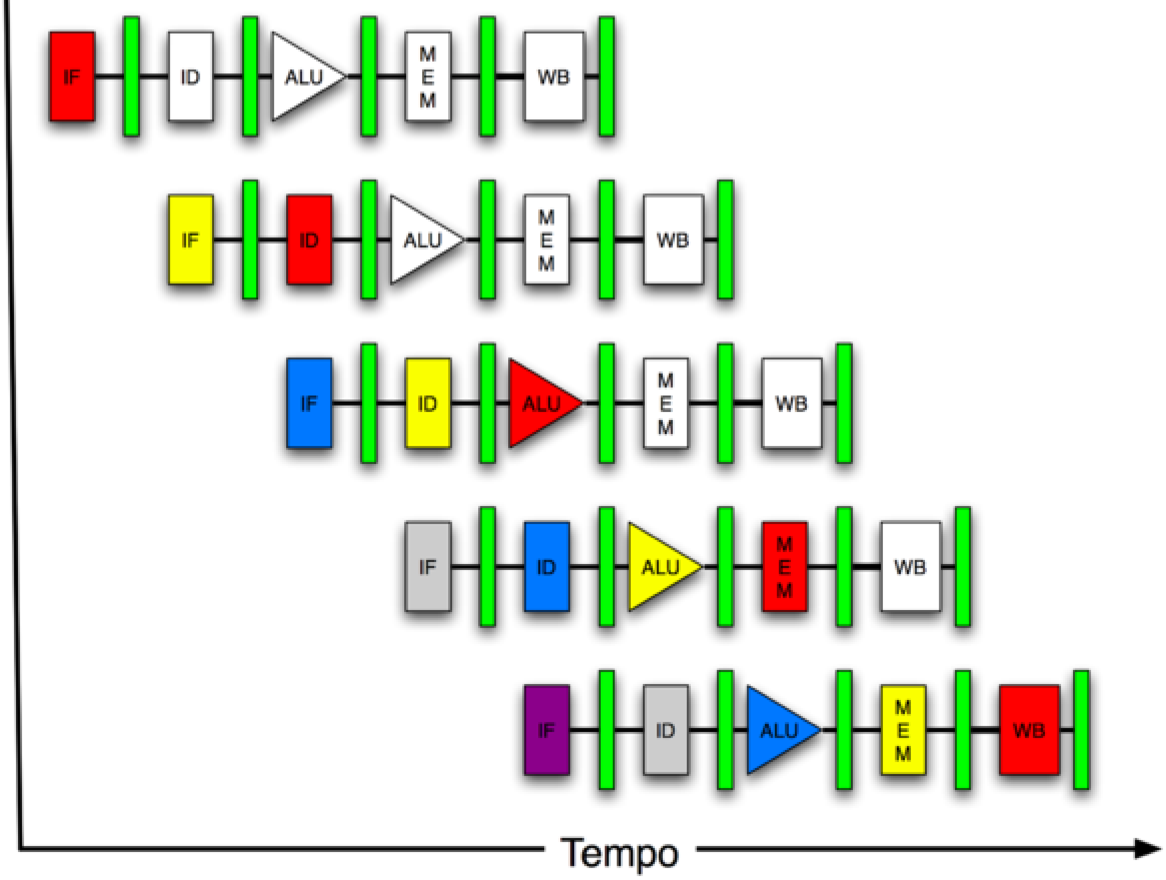
Appendix: Pipelining Hardware

- Our resultant processor design will look similar to this:



Appendix: Pipelining Hardware

- If we have 5 instructions, we can show them in our pipeline using different colors. In the diagram below, white corresponds to a NOP, and the different colors correspond to other instructions in the pipeline. Each stage, the instructions shift forward through the pipeline.


IF
ID
ALU
M E M
WB
IF
ID
ALU
M E M
WB
IF
ID
ALU
M E M
WB
IF
ID
ALU
M E M
WB
IF
ID
ALU
M E M
WB
Tempo