COMP 212 Fall 2008

Lecture 11

Control Unit, Pipeline

Overview

- **Control Unit:**
  - CPU operations can be broken into smaller time scale "micro-operations"
  - Control unit co-ordinates these operations
  - Can be implemented as circuits, or micro-programmed.
- **Pipeline:**
  - Different instruction's Micro-operations can overlap each other
  - Achieve parallelism by having a "pipeline" of operations.

Control Unit

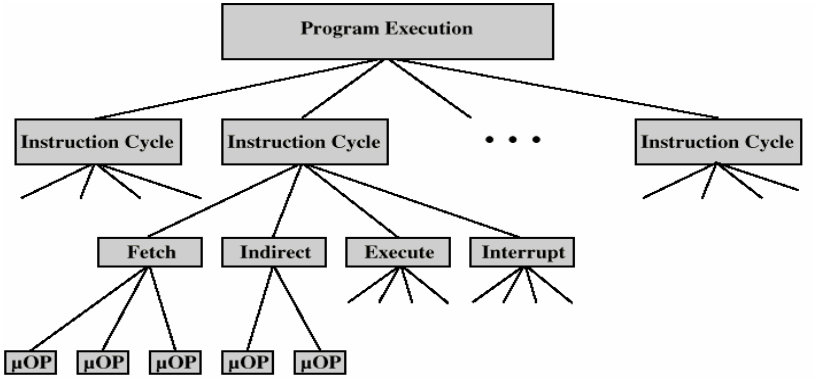
(refer to chapter 16)

Processor Functional Spec

- **Instruction set is still a high level definition, in more detail, we need to know:**
  - Operations (opcode)
  - Addressing modes
  - Registers
  - I/O interface
  - Memory module interface
  - Interrupt handling structure

Instruction Cycle Micro-Operations

- A computer executes a program
- A program has many instruction cycles
- Each instruction cycle has a number of steps (pipelines) called micro-operations



Fetch Cycle – Registers involved

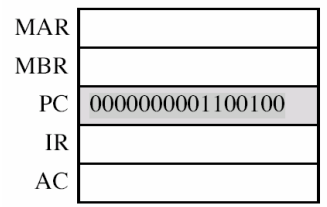
- **Memory Address Register (MAR)**
  - Connected to address bus
  - Specifies address for read or write op
- **Memory Buffer Register (MBR)**
  - Connected to data bus
  - Holds data to write or last data read
- **Program Counter (PC)**
  - Holds address of next instruction to be fetched
- **Instruction Register (IR)**
  - Holds last instruction fetched

Fetch Cycle Micro-Ops

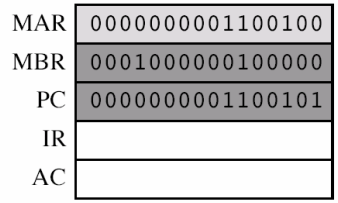
$$t1: MAR \leftarrow (PC)$$
$$t2: MBR \leftarrow Memory$$
$$t3: PC \leftarrow (PC) + I$$
$$IR \leftarrow (MBR)$$

- Address of next instruction is in PC
- Address (MAR) is placed on address bus
- Control unit issues READ command
- Result (data from memory) appears on data bus
- Data from data bus copied into MBR
- PC incremented by 1 (in parallel with data fetch from memory)
- Data (instruction) moved from MBR to IR
- MBR is now free for further data fetches

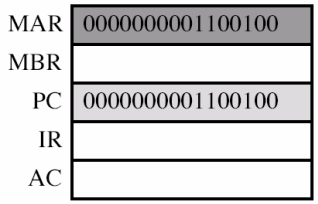
Fetch Cycle Register States



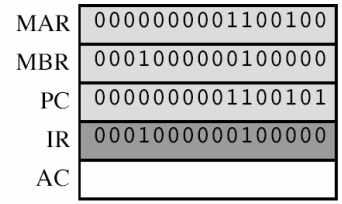
(1)



(3)



(2)



(4)

Rules for Clock Cycle Grouping

- **Proper sequence must be followed**
  - MAR <- (PC) must precede MBR <- (memory)
- **Conflicts must be avoided**
  - Must not read & write same register at same time
  - MBR <- (memory) & IR <- (MBR) must not be in same cycle
- **Also: PC <- (PC) +1 involves addition**
  - Use ALU
  - May need additional micro-operations

Indirect Cycle

$$t1: MAR \leftarrow (IR(Addr))$$
$$t2: MBR \leftarrow Memory$$
$$t3: IR(Addr) \leftarrow (MBR(Addr))$$

- t1: Need to load data addr from memory
- t2:MBR contains an address
- t3: IR is now in same state as if direct addressing had been used

Interrupt Cycle

$$t1: MBR \leftarrow (PC)$$
$$t2: MAR \leftarrow SavePcToAddr$$
$$PC \leftarrow IsrAddr$$
$$t3: Memory \leftarrow (MBR)$$

- This is a minimum
  - Steps t1, t2: save current PC addr
  - Step 3: store MBR, which is the old value of PC

Execute Cycle

- **Fetch and Interrupt Cycles's micro-ops are fixed**
- **Micro-ops for execution cycle is different for each instruction**

Execute Cycle (ADD)

- **ADD R1,X - add the contents of location X to Register 1 , result in R1**
- **t1: MAR <- IR(X)**
- **t2: MBR <- (memory)**
- **t3: R1 <- R1 + (MBR)**
- **Note no overlap of micro-operations**

Execute Cycle (ISZ)

- **ISZ X - increment and skip if zero**
  - t1: MAR <- (IR(X))
  - t2: MBR <- (memory)
  - t3: MBR <- (MBR) + 1
  - t4: memory <- (MBR)
  - if (MBR) == 0 then PC <- (PC) + 1
- **Notes:**
  - if is a single micro-operation
  - Micro-operations done during t4

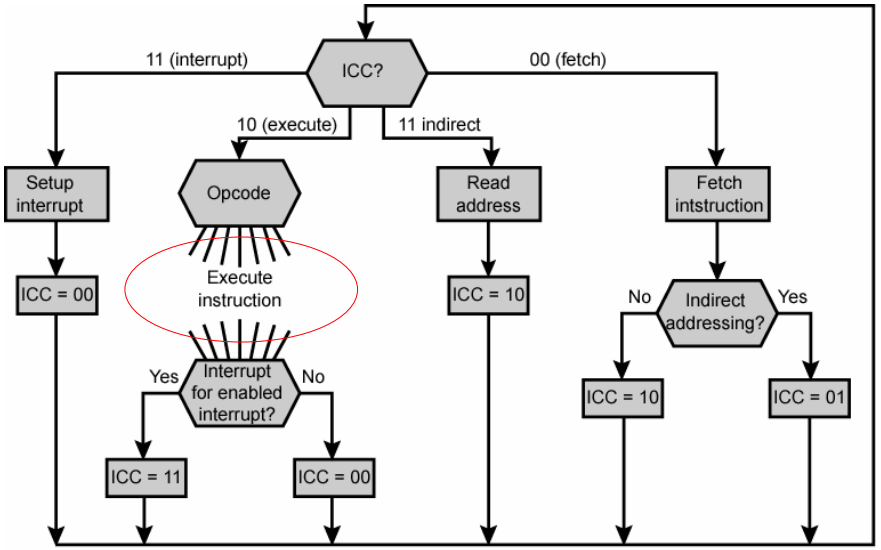
Execute Cycle (BSA)

- **BSA X - Branch and save address**
  - Address of instruction following BSA is saved in X
  - Execution continues from X+1
  - t1: MAR <- (IR(X))
  - MBR <- (PC)
  - t2: PC <- (IR(X))
  - memory <- (MBR)
  - t3: PC <- (PC) + 1

Instruction Cycle

- **Each phase decomposed into sequence of elementary micro-operations**
- **E.g. fetch, indirect, and interrupt cycles**
- **Execute cycle**
  - One sequence of micro-operations for each opcode
  - Can be different with different number of micro-ops
- **Need to tie sequences together**
- **Assume a new 2-bit register**
  - Instruction cycle code (ICC) designates which part of cycle processor is in
    - » 00: Fetch
    - » 01: Indirect
    - » 10: Execute
    - » 11: Interrupt

Flowchart for Instruction Cycle



Processor Control

- **How to implement controls of CPU ?**
  - What are the basic elements in CPU ?
  - Micro-operations involved ?
  - Functions and logic/circuits that implement these micro-operations and sequences

Basic Elements of Processor

- **ALU**
  - Operates on registers with Arithmetic & Logic functions
- **Registers**
  - Internal CPU storage
- **Internal data paths**
  - Connects registers, ALU.
- **External data paths**
  - Connect registers and memory
- **Control Unit**

Types of Micro-operation

- **Transfer data between registers**
  - e.g, move $R1, $R2
- **Transfer data from register to external**
  - E.g. sw $R1, $ra
- **Transfer data from external to register**
  - lw r1, $r0
- **Perform arithmetic or logical ops**
  - Operates on registers

Control Unit Implementation

Functions of Control Unit

- **Sequencing**
  - Causing the CPU to step through a series of micro-operations
- **Execution**
  - Causing the performance of each micro-op
- **This is done using Control Signals**

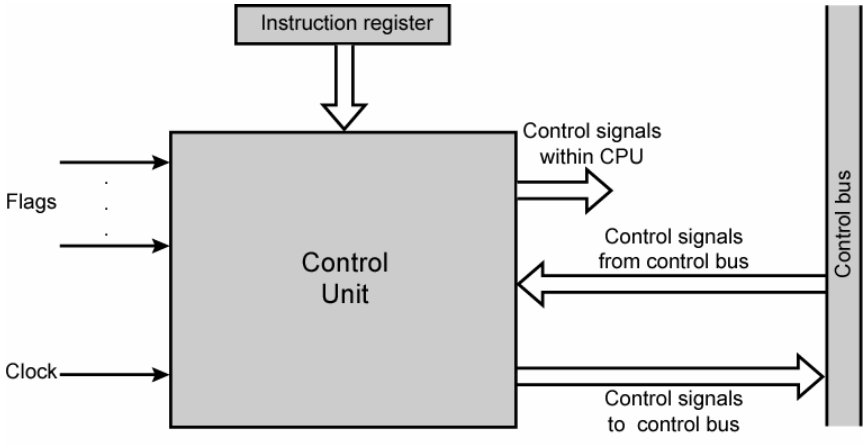
Control Signal Input

- **Clock**
  - One micro-instruction (or set of parallel micro-instructions) per clock cycle
- **Instruction register**
  - Op-code for current instruction
  - Determines which micro-instructions are performed
- **Flags**
  - State of CPU
  - Results of previous operations
- **From control bus**
  - Interrupts
  - Acknowledgements

Control Signals Output

- **Within CPU**
  - Cause data movement among registers
  - Activate specific functions, eg, ALU
- **Via control bus**
  - To memory
  - To I/O modules

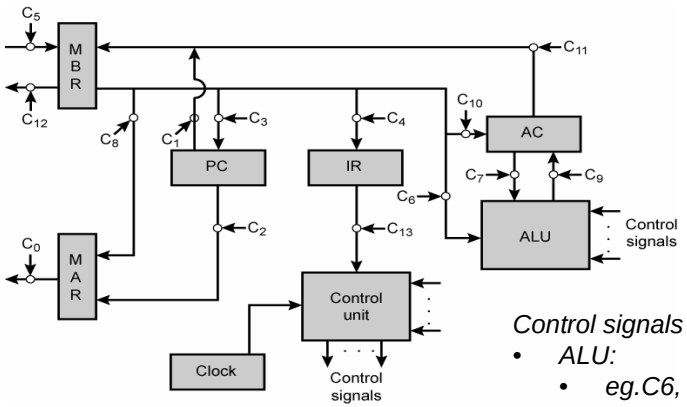
Model of Control Unit



Example Control Signal Sequence - Fetch

- **MAR <- (PC)**
  - Control unit activates signal to open gates between PC and MAR
- **MBR <- (memory)**
  - Open gates between MAR and address bus
  - Memory read control signal
  - Open gates between data bus and MBR

Control Signal Circuits Example



Control signals for:

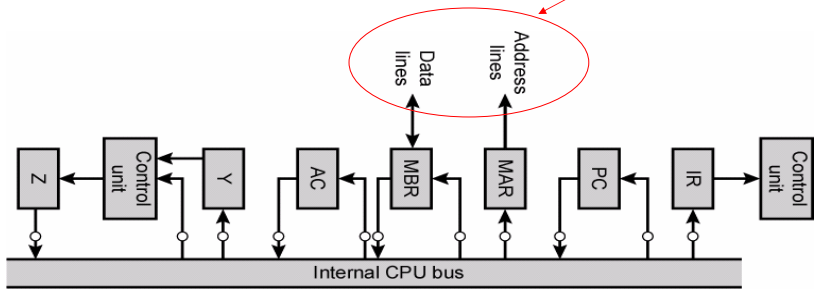
- *ALU:*
  - *eg.C6, C7, C9, C10*
- *Internal Data paths:*
  - *Eg C6*
- *System Bus*
  - *Eg C5, C12*

- **Micro-Ops and Control signals Fetch, Indirect and Int**

Micro-operations	Timing	Active Control Signals
Fetch:	t_1: MAR ← (PC)	C_2
	t_2: MBR ← Memory PC ← (PC) + 1	C_5, C_R
	t_3: IR ← (MBR)	C_4
Indirect:	t_1: MAR ← (IR(Address))	C_8
	t_2: MBR ← Memory	C_5, C_R
	t_3: IR(Address) ← (MBR(Address))	C_4
Interrupt:	t_1: MBR ← (PC)	C_1
	t_2: MAR ← Save-address PC ← Routine-address	
	t_3: Memory ← (MBR)	C_{12}, C_W

Internal Bus Organization

- Usually a single internal bus
- Gates control movement of data onto and off the bus
- Control signals control data transfer to and from external systems bus
- Temporary registers needed for proper operation of ALU



Control Unit Implementation

- **Clocks, Flags, control bus signals:**
  - Each bit is clearly defined, can be implemented by logics
- **Opcode from IR:**
  - Need to generate a sequence of control signals depending on opcode
  - Implemented by a decoder on Opcode

Hardwired Implementation (1)

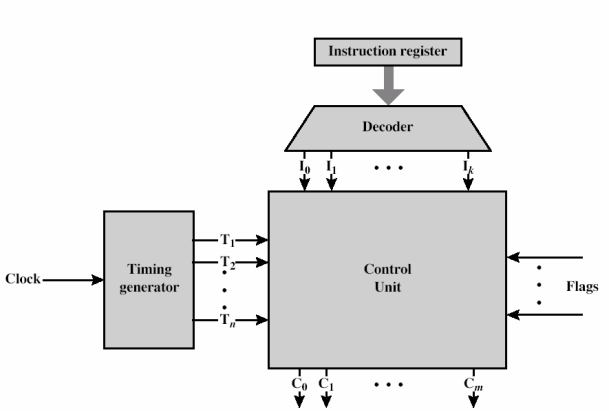
- **Control unit inputs**
- **Flags and control bus**
  - Each bit means something
- **Instruction register**
  - Op-code causes different control signals for each different instruction
  - Unique logic for each op-code
  - Decoder takes encoded input and produces single output
  - n binary inputs and 2^n outputs

Hardwired Implementation (2)

- **Clock**
  - Repetitive sequence of pulses
  - Useful for measuring duration of micro-ops
  - Must be long enough to allow signal propagation
  - Different control signals at different times within instruction cycle
  - Need a counter with different control signals for t1, t2 etc.

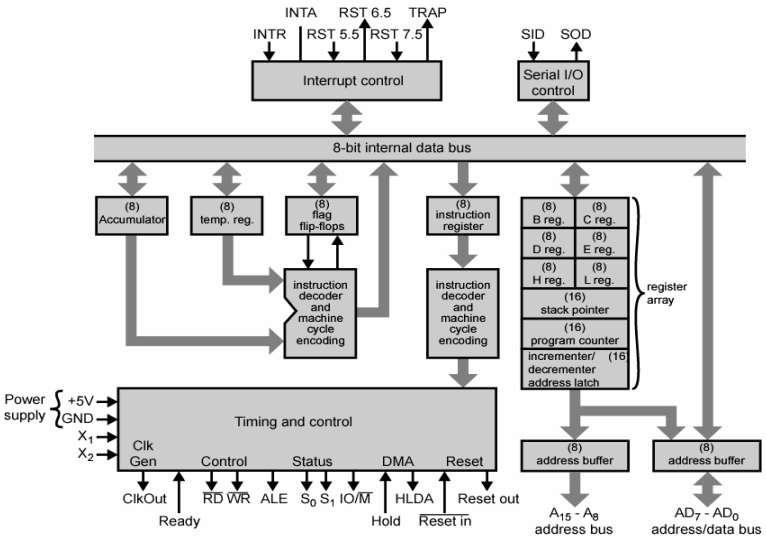
Control Unit Implementation

- **Clocks, Flags, control bus signals:**



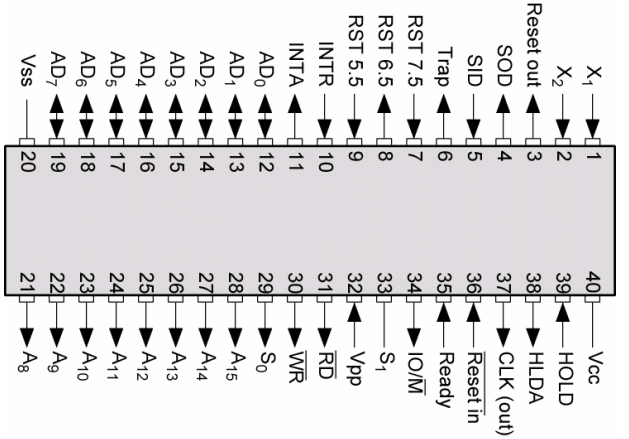
Control Unit Example (Informational)

Intel 8085 CPU Block Diagram

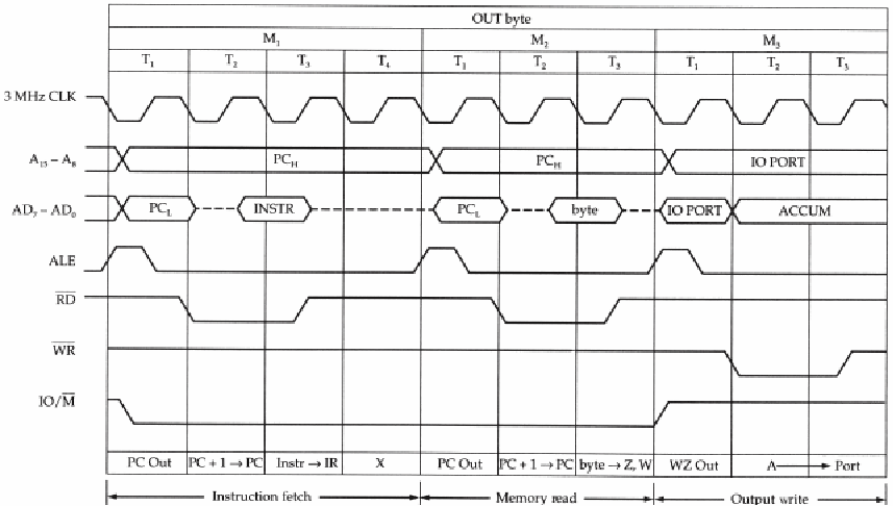


Intel 8085

Intel 8085 control unit



Intel 8085 Timing

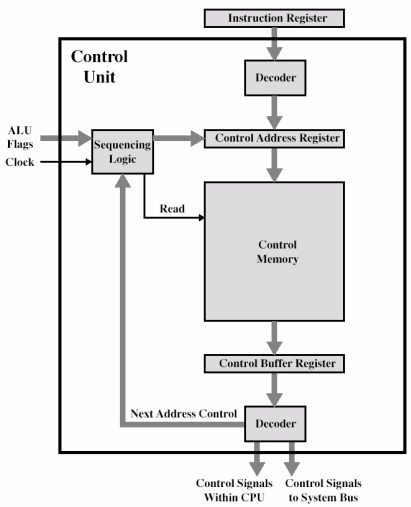


Problems With Hard Wired Designs

- Complex sequencing & micro-operation logic
- Difficult to design and test
- Inflexible design
- Difficult to add new instructions
- Solution: Micro-program

Micro-Programmed Control

- **A brief intro:**
  - No hardwiring of control signals to the input Flag/Clock/Instruction
  - Load control signals from control memories
  - Controlled by control addr reg



Control Unit Summary

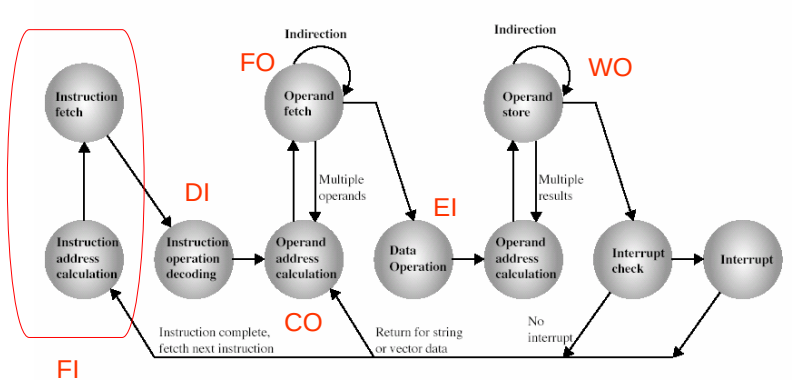
- Control unit handles CPU operations and communications with Registers and Memory
- Input: Instructions, Clock, and Flags
- Output: control signals and sequences that allow instruction to be fetched, operands loaded and instruction executed correctly
- Can be implemented as hardwired circuits
- Can also be implemented as micro-programmed controls.

Pipeline

Operation Cycles of Instructions

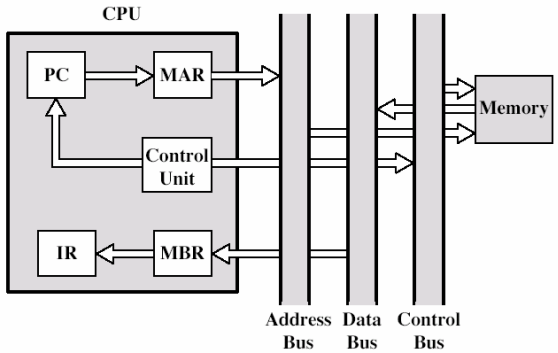
- **A typical instruction cycle will typically have:**
  - Fetch Instruction (FI): load instruction
  - Decode Instruction (DI): understand what to do
  - Calculate Operands (CO): calculate the effective address of operands
  - Fetch Operands (FO): fetch operands from memory
  - Execute Instruction (EI): generate necessary control signals and sequences to finish operation on operands in registers
  - Write Operands (WO): write operand from register to memory

- **State Diagram of Instruction Cycle**



Data Flows

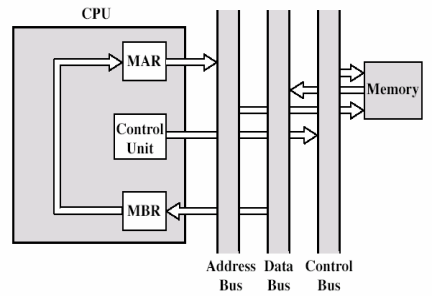
- **Fetch Cycle Data Flow:**



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

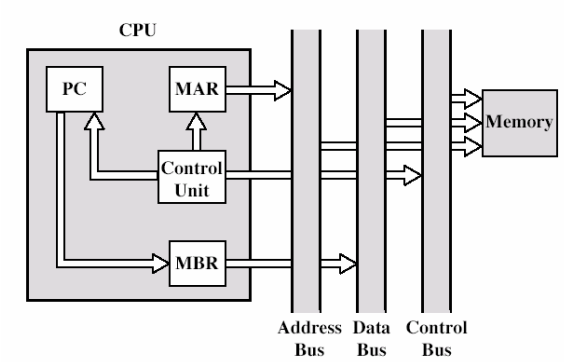
Data Flows

- **Indirect Cycles:**
  - Load operands that are not in registers
  - Addr of operand in MBR => MAR



Data Flows

- **Interrupt Handling Cycles:**
  - Save current PC to MBR, and then to mem (e.g, pointed by sp)

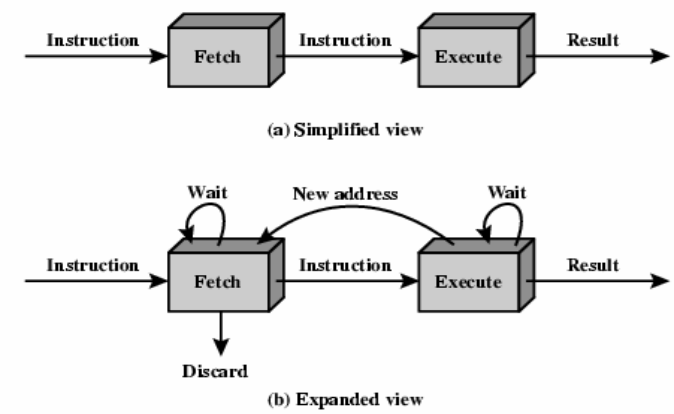


How to speed up instruction execution ?

- **Question:**
  - Do we have to wait for all cycles to complete before starting the next instruction ?
- **Answer:**
  - No, we can do better
  - It is called Pipeline

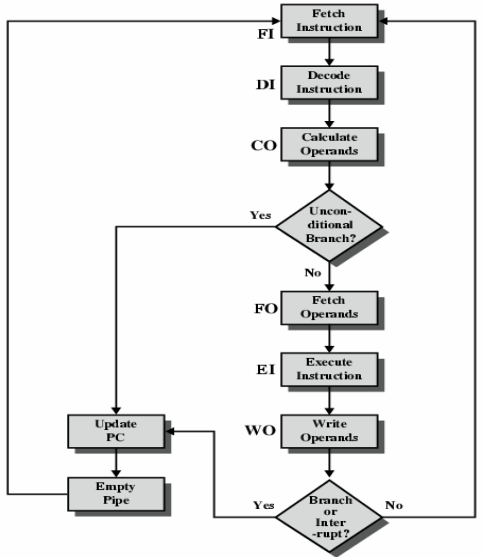
Simplified Two Stage Instruction Pipeline

- **Allows overlapping of fetch and execute cycles**

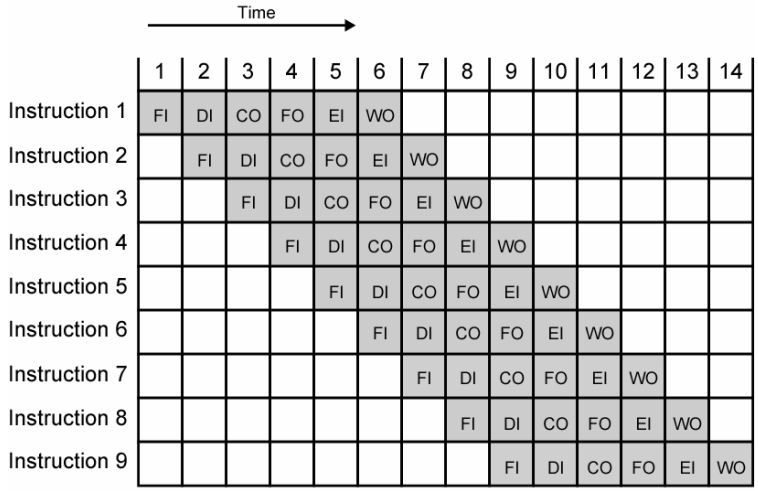


Instruction Micro-Operations

- **An 6-stage pipeline**
  - Execution takes longer than fetch
  - Break up execution into sub-cycles, i.e, DI, CO, FO, EI, WO.
  - Allow overlapping, or pre-fetch the command
  - Branch : may have to re-fetch the correct instruction



Instruction Pipeline – no branching



Speedup: 9x6=54 (no pipeline) vs 14 (pipelined) time slots.

Pipeline efficiency

- **Non-uniform pipeline stages**
  - Involves waiting at different stages
- **Conditional branch**
  - Pre-fetched instructions become invalid
- **Data dependency**
  - Instruction k writes to memory, instruction k+1 reads the data
  - Then need to wait instruction k's WO to finish,
- **The more stages the better ?**
  - No, additional data moving slows down the execution, plus circuits complexity

Pipeline Harzards

- **Structual:**

  Structural hazzards occur when the same functional unit is needed to be used

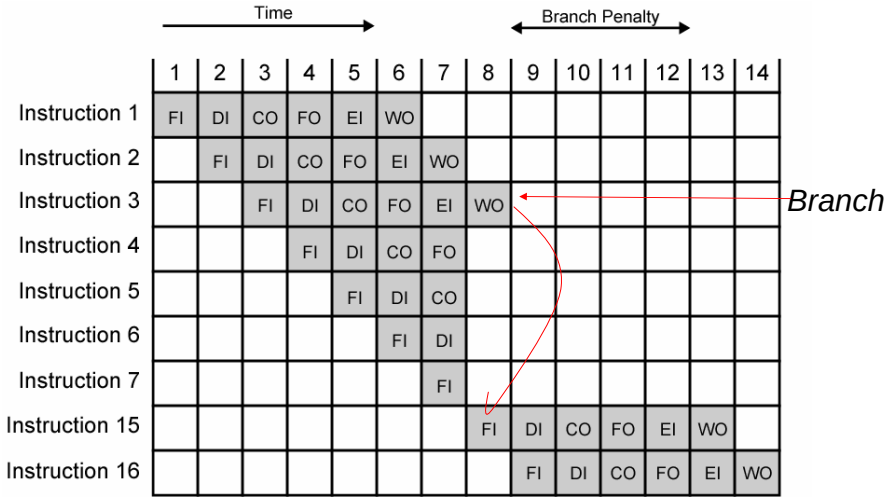
  by two different instructions at the same time in the same cycle.
- **Control:**

  Branches and jumps (subroutine calls) disrupt the sequence of instructions being issued and may result in a pipeline stall. Early branch scheduling and branch prediction are used to minimize the number of stalls.
- **Data:**

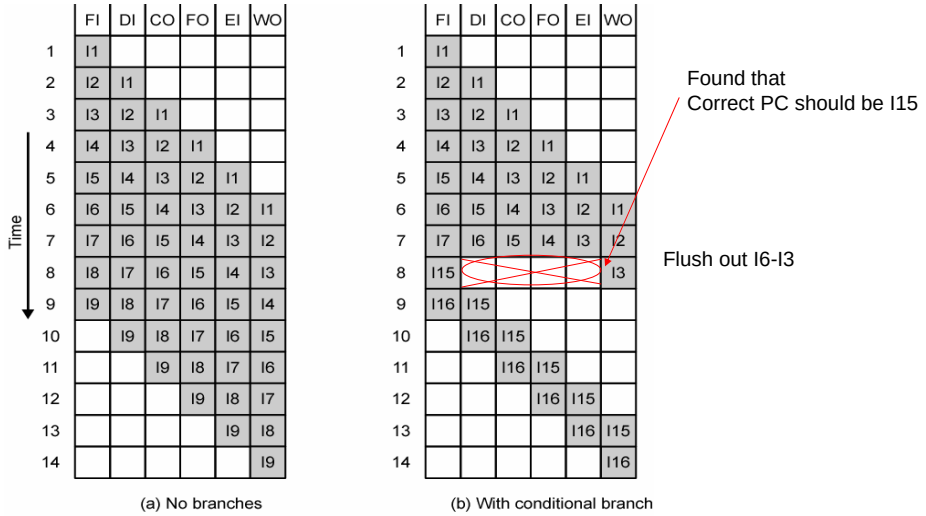
  Data hazzards occur when an instruction reads from a register before an earlier instruction has written the expected value into the register (WAR),

Conditional branching

- **The correct PC address is runtime dependent**



Alternative Pipeline View



Pipeline Efficiency Analysis

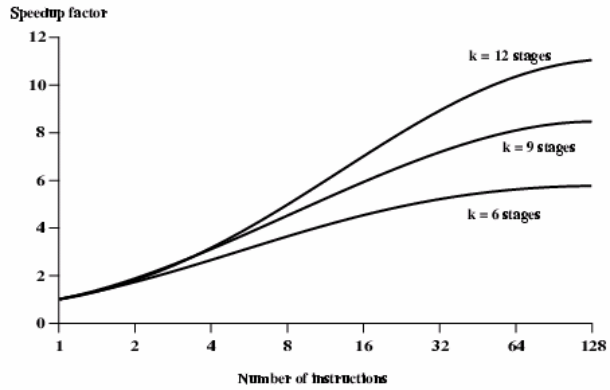
- K-stage pipeline, n instructions, execution time:

$$T_{k,n} = [k + (n-1)]\tau$$

- Speed up factor as function of stages

$$S_k = \frac{T_{1,n}}{T_{k,n}} = \frac{nk\,\tau}{[k + (n-1)]\tau} = \frac{nk}{[k + (n-1)]}$$

Speedup



Dealing with Branches

- Pipeline efficiency depends on a steady stream of instructions that fills up the pipeline
- Conditional branching is a major drawback for efficiency
- Can be deal with by:
  - 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
- Leads to bus & register contention
- Multiple branches lead to further pipelines being needed

Prefetch Branch Target

- Target of branch is prefetched in addition to the instructions following branch (PC+1)
- Keep target until branch is executed
- Used by IBM 360/91

Loop Buffer

- Very fast memory in CPU
- Maintained n most recently fetched instructions
- Check buffer before fetching from memory
- Very good for small loops or jumps
  - E.g. typical IF THEN, IF THEN ELSE sequences
  - Like cache for instructions.
  - Used in supercomputer like CRAY-1

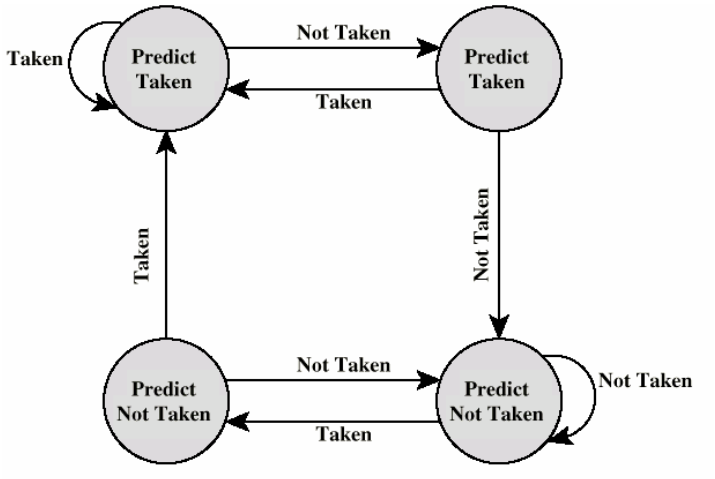
Branch Prediction – Static Solutions

- **Predict never taken**
  - Assume that jump will not happen
  - Always fetch next instruction
  - 68020 & VAX 11/780
- **Predict always taken**
  - Assume that jump will happen
  - Always fetch target instruction
- **Predict by opcode**
  - By collecting stats on different opcode w.r.t. branching
  - Correct rate > 75%

Branch Prediction – Dynamic, Runtime Based

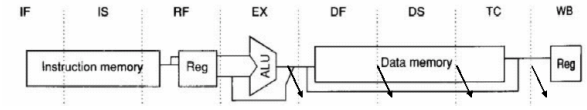
- **Taken/Not taken switch**
  - Use 1 or 2 bits to record taken/not taken history
  - Good for loops
- **Branch history table**
  - Based on previous history
  - Good for loops

Branch Prediction State Diagram



Examples of Pipeline designs

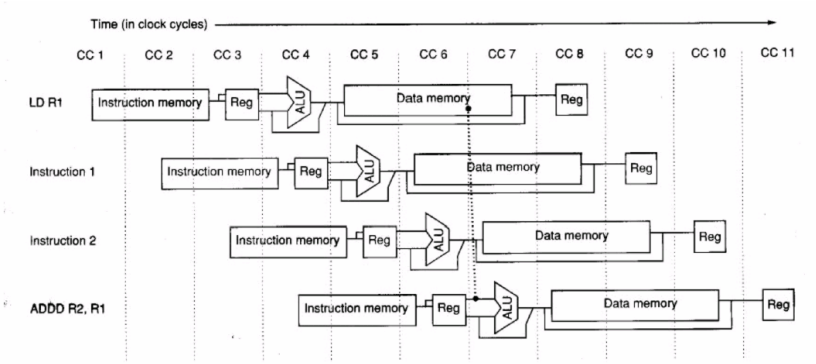
MIPS-Microprocessor w/o Interlocked Pipeline Stages



- IF—First half of instruction fetch; PC selection actually happens here, together with initiation of instruction cache access.
- IS—Second half of instruction fetch, complete instruction cache access.
- RF—Instruction decode and register fetch, hazard checking, and also instruction cache hit detection.
- EX—Execution, which includes effective address calculation, ALU operation, and branch-target computation and condition evaluation.
- DF—Data fetch, first half of data cache access.
- DS—Second half of data fetch, completion of data cache access.
- TC—Tag check, determine whether the data cache access hit.
- WB—Write back for loads and register-register operations.

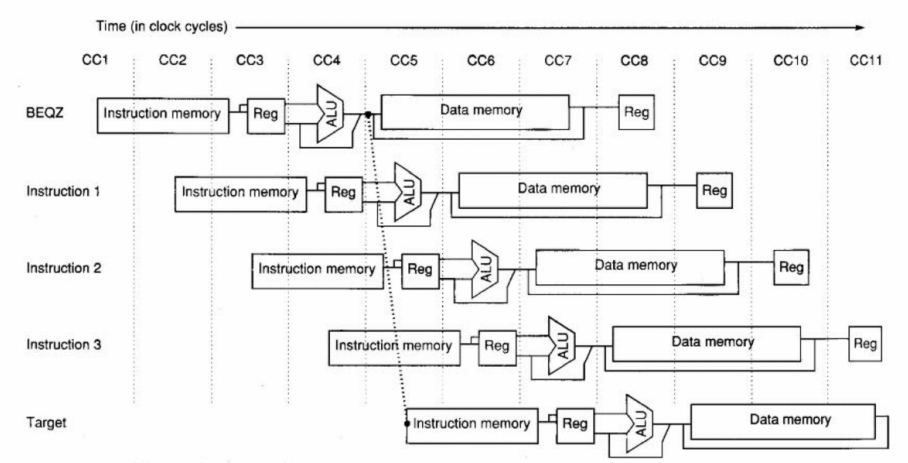
Delay from load stalls

- **2 cycle delay for load data:**
  - Data available only at the end of DS (cc 6 for instruction 1)



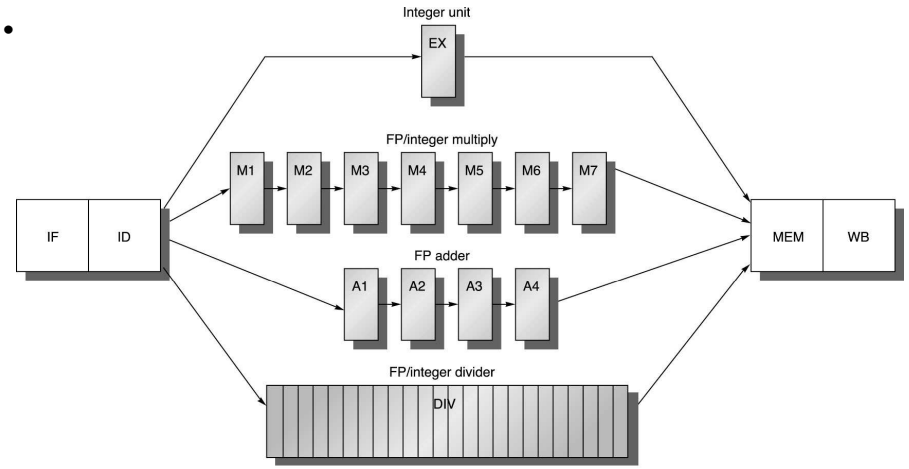
3-cycle delay for branch

- **Target addr available after EX**



Floating point op cycles

-



Pipeline Summary

- **Pipeline is ILP -Instruction Level Parallelism**
  - Could potentially achieve k time speed up for k-stage pipelines
- **Pipeline Hazards:**
  - Structural: two micro-ops requires the same circuits in the same cycle
  - Control: target branch PC not known until execution
  - Data: successive instructions read the output of previous instruction