Revised: 5/9/2024

Processor Architecture II

Michael C. Hackett

Assistant Professor, Computer Science

COMMUNITY COLLEGE OF PHILADELPHIA

Lecture Topics

- Pipelining
- Pipeline Hazards
- Interrupts

Pipelining

- **Pipelining** allows multiple instructions to overlap in execution
  - In other words, fetch-decode-execute instruction cycles happen in parallel rather than one-at-a-time in a single-cycle processor
- Higher throughput than single-cycle processors
- Modern processors use pipelining
- Processors that *do not* use pipelining are called “scalar processors”
- Processors that *do* use pipelining are called “superscalar processors”

Pipelining

- Single-cycle processors complete one instruction cycle at a time.
- Typical instruction cycle:
  - **Fetch**
    - The next instruction is fetched from the instruction memory, at the memory address currently in the program counter
    - The instruction stored into the instruction register
  - **Decode**
    - The instruction stored in the instruction register is decoded
  - **Execute**
    - The control unit signals the relevant functional units of the processor to perform the instruction.
  - (Repeat…)

Pipelining

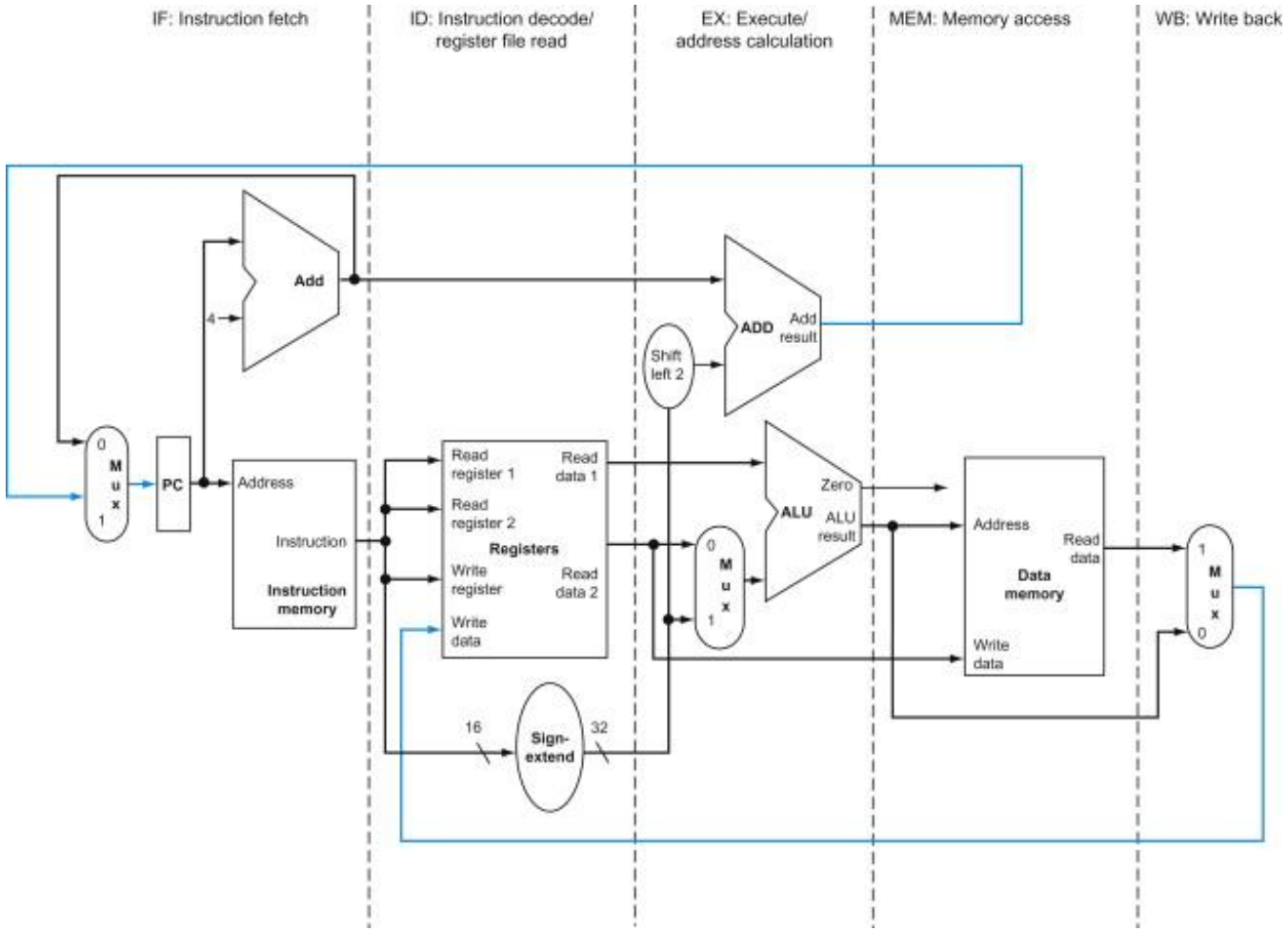
- Instruction cycles between different processors may vary:
  - MIPS instruction cycle:
    - **Fetch (IF)**
      - Fetch instruction from memory.
    - **Decode (ID)**
      - Read registers while decoding the instruction. The regular format of MIPS instructions allows reading and decoding to occur simultaneously.
    - **Execute (EX)**
      - Execute the operation or calculate an address.
    - **Memory (MEM)**
      - Access an operand in data memory.
    - **Write Back (WB)**
      - Write the result into a register.

Pipelining

- The division of an instruction into five stages means a five-stage pipeline
  - Further, up to five instructions will be in execution during any single clock cycle.

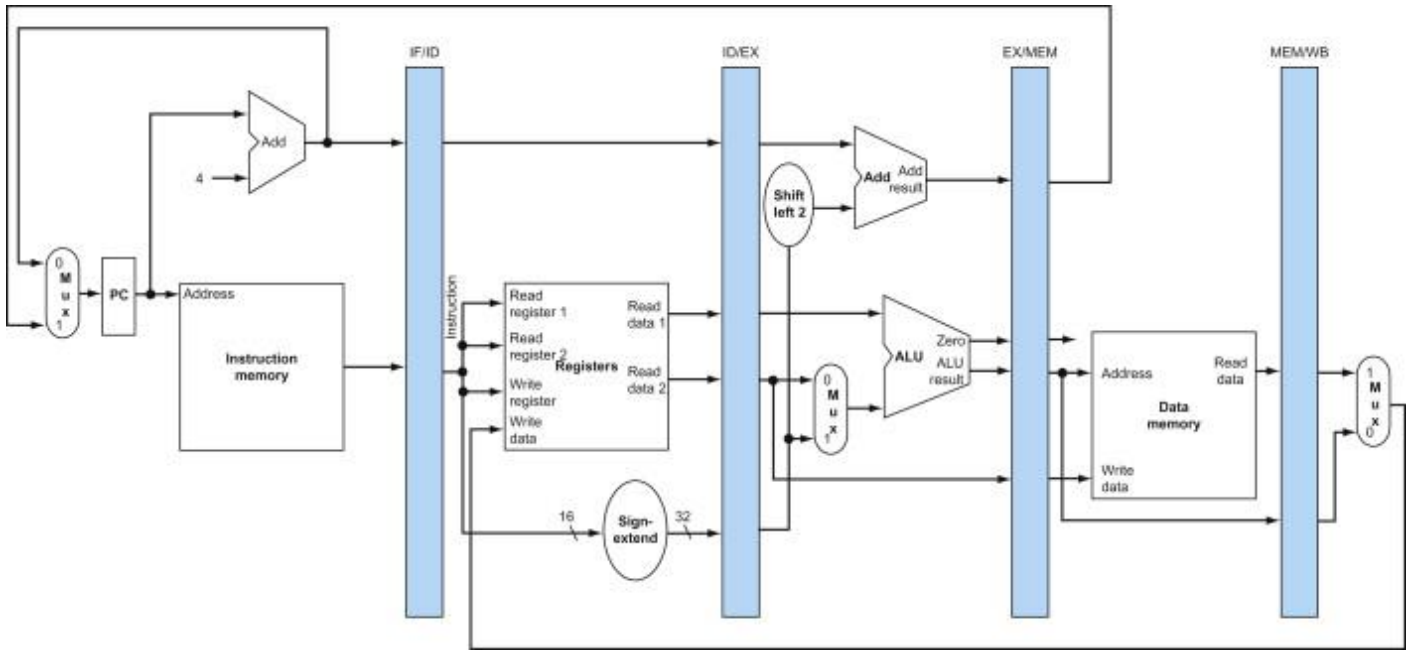
- The datapath must separates into five pieces, with each piece named corresponding to a stage of instruction execution.

Pipelining


IF: Instruction fetch
ID: Instruction decode/ register file read
EX: Execute/ address calculation
MEM: Memory access
WB: Write back
Add
4
M u x
PC
Address
Instruction
Instruction memory
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
ADD
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
16
Sign-extend
32

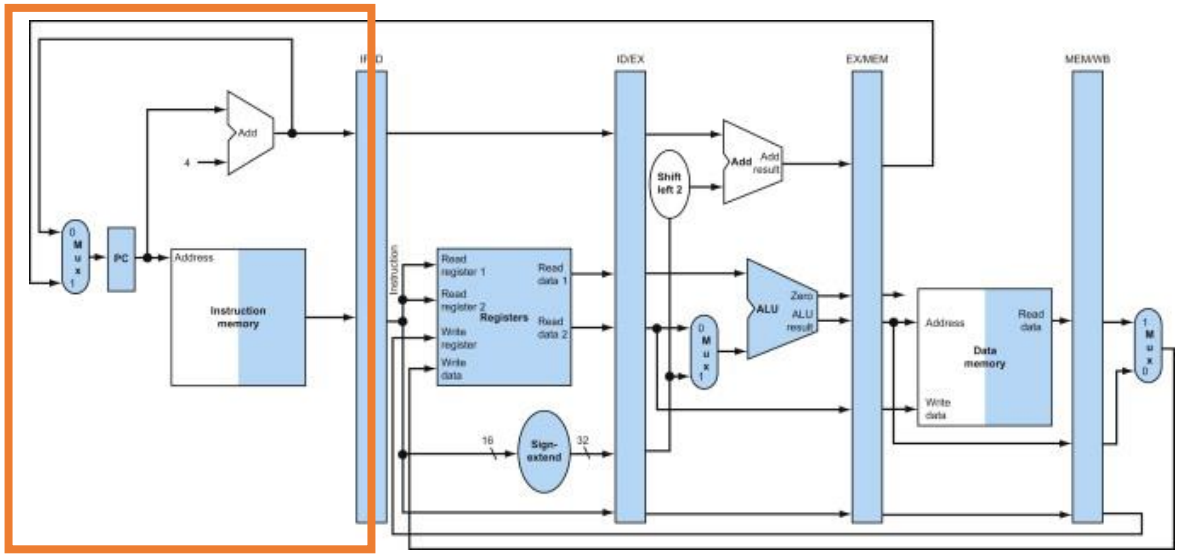
Pipelining

- The figure below shows the pipelined datapath with the pipeline registers highlighted.
  - All instructions advance during each clock cycle from one pipeline register to the next.



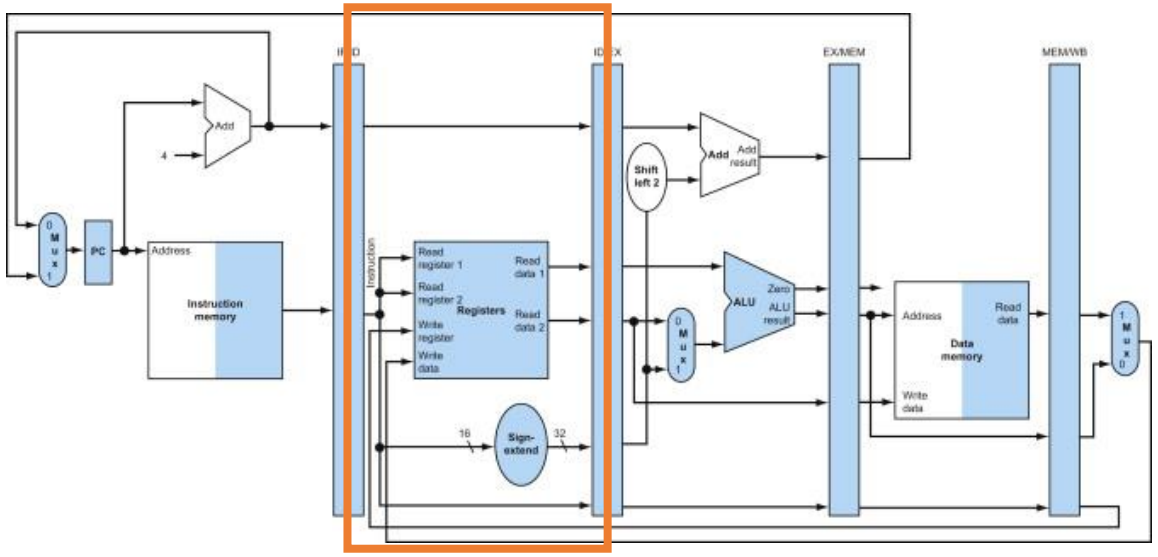
Pipelining

- The instruction is read from memory using the address in the PC
  - Placed in the IF/ID pipeline register.
  - PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle.



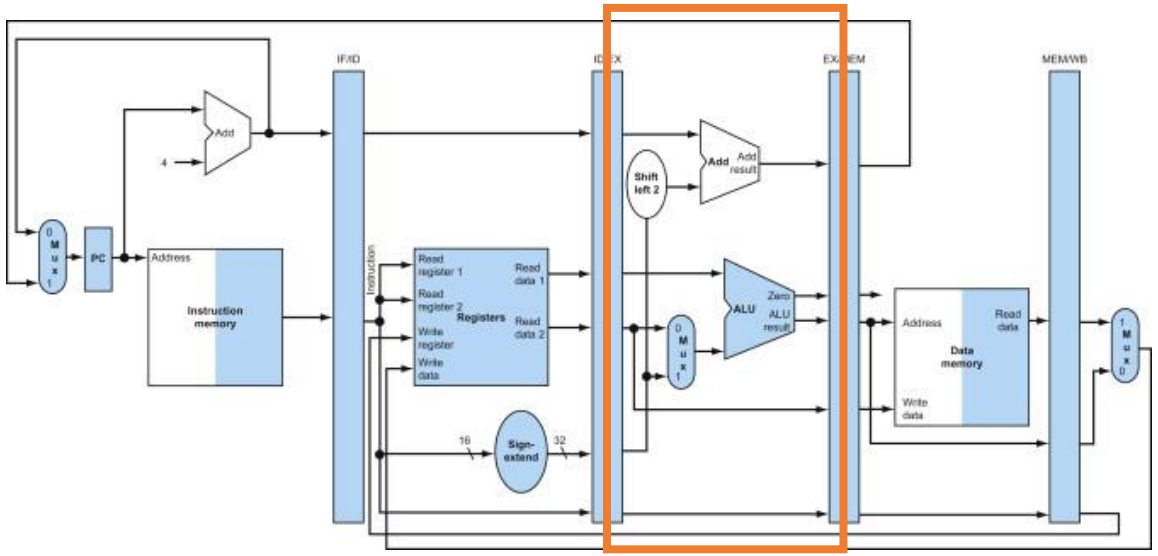
Pipelining

- The instruction portion of the IF/ID pipeline register supplies the 16-bit immediate field, sign-extended to 32 bits, and the register numbers to read the two registers.



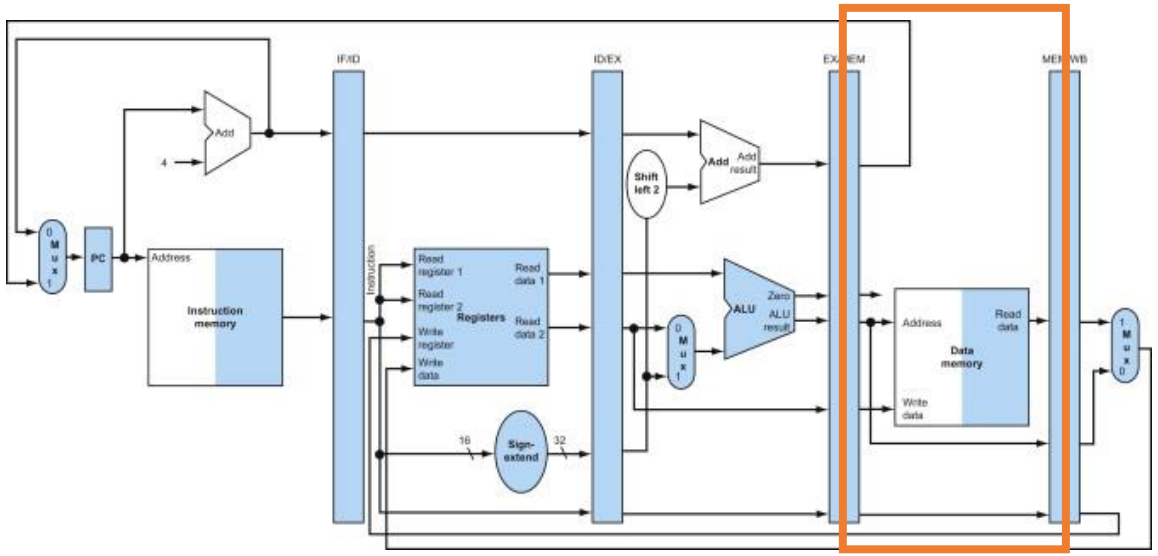
Pipelining

- The load instruction reads the contents of register 1 and the sign-extended immediate from the ID/EX pipeline register
  - Adds them using the ALU; The sum is placed in the EX/MEM pipeline register.



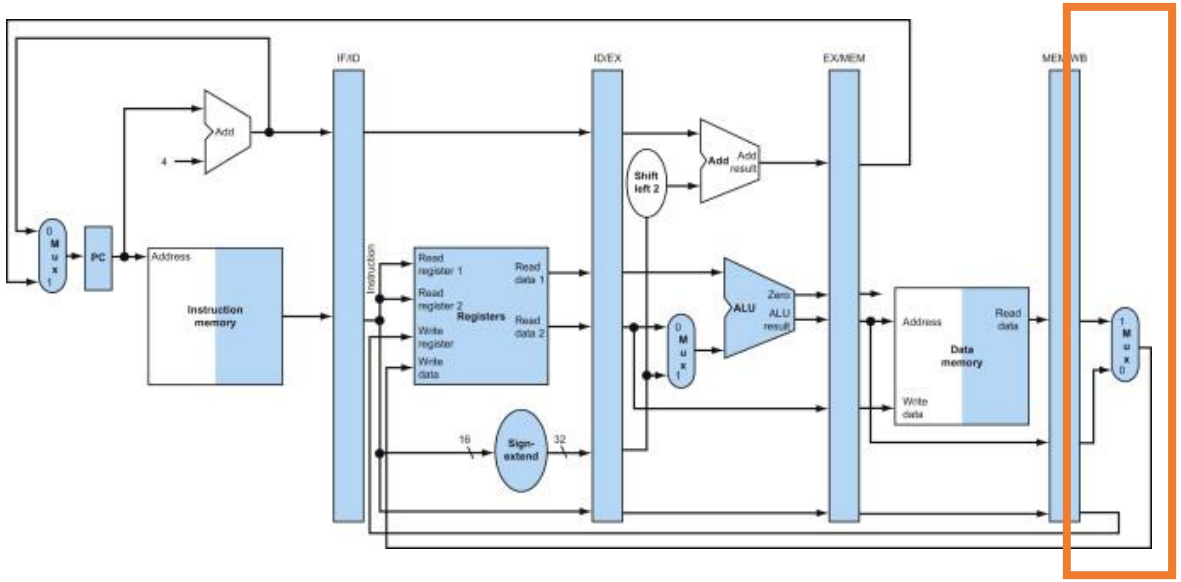
Pipelining

- The load instruction reads the data memory using the address from the EX/MEM pipeline register and loads the data into the MEM/WB pipeline register.



Pipelining

- The data from the MEM/WB pipeline register is read and written into the register file



Pipelining

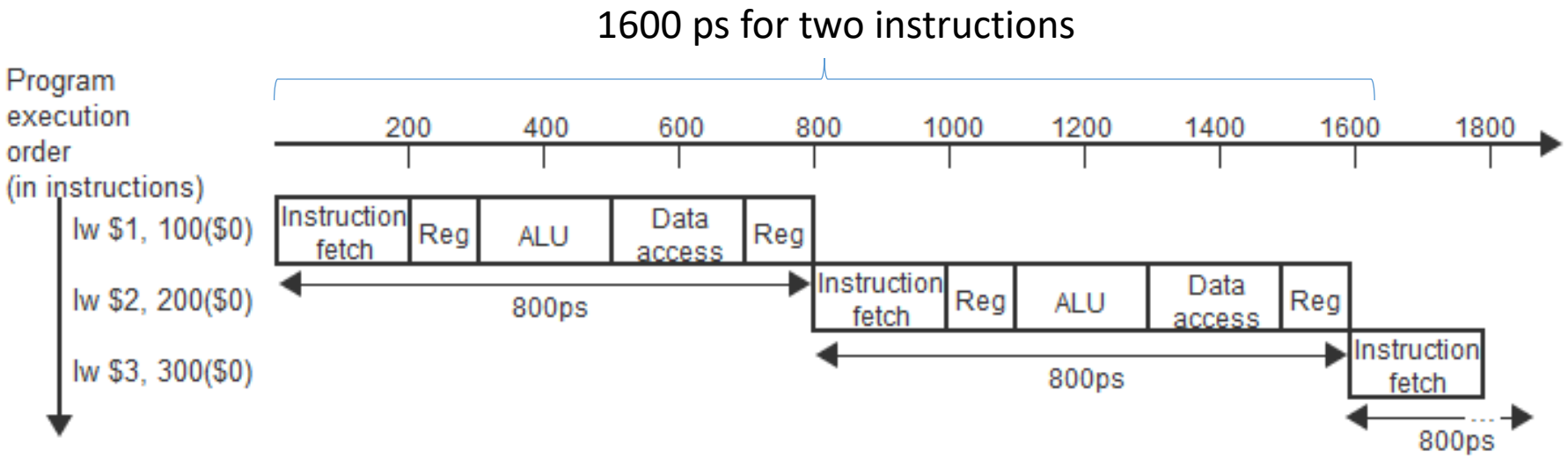
- Pipelining is a technique that exploits **parallelism**.
  - As the name suggests, parallelism refers to increasing processor performance by performing operations simultaneously.

- Pipelining is **instruction-level parallelism** and is abstracted away from view of programmers and compilers.
  - As far as they can tell, the processor executes instructions sequentially.

Pipelining

- Single-cycle execution

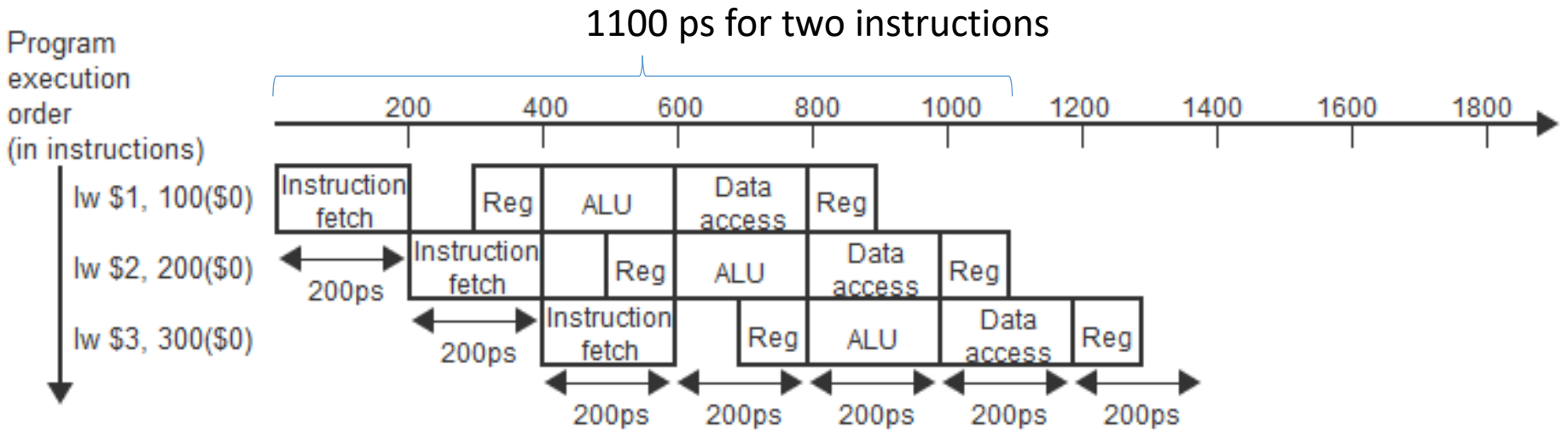
Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps



Pipelining

- Pipelined/multi-cycle execution

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps



Pipelining

- At first, the higher throughput may not appear to offer that much of an advantage (only a 500 ps improvement)

- Consider 1000000 `lw` instructions executed in the two previous examples
  - Time between the start of each instruction
    - Single-cycle: 800ps
    - Multi-cycle: 200ps
  - Total Time
    - Single-cycle: 800000000ps (.0008s)
    - Multi-cycle: 200000000ps (.0002s)

- The multi-cycle processor was 4 times faster than single-cycle
  - Improves performance by increasing throughput, not by decreasing the execution time of an instruction

Pipelining

- One technique to further increase performance is *superpipelining,* where instructions are split into many more stages than MIPS's five-stage instruction cycle
  - Upwards of 12 or more stages

- Another technique is, depending on how the logic for scheduling instructions is designed, its possible for two or more instructions to execute within the *same* clock cycle.
  - If the instructions each require different resources from the others, then the processor can execute those instructions without any conflicts or *hazards*.

Pipeline Hazards

- A **hazard** is when an instruction cannot execute in during the next clock cycle.

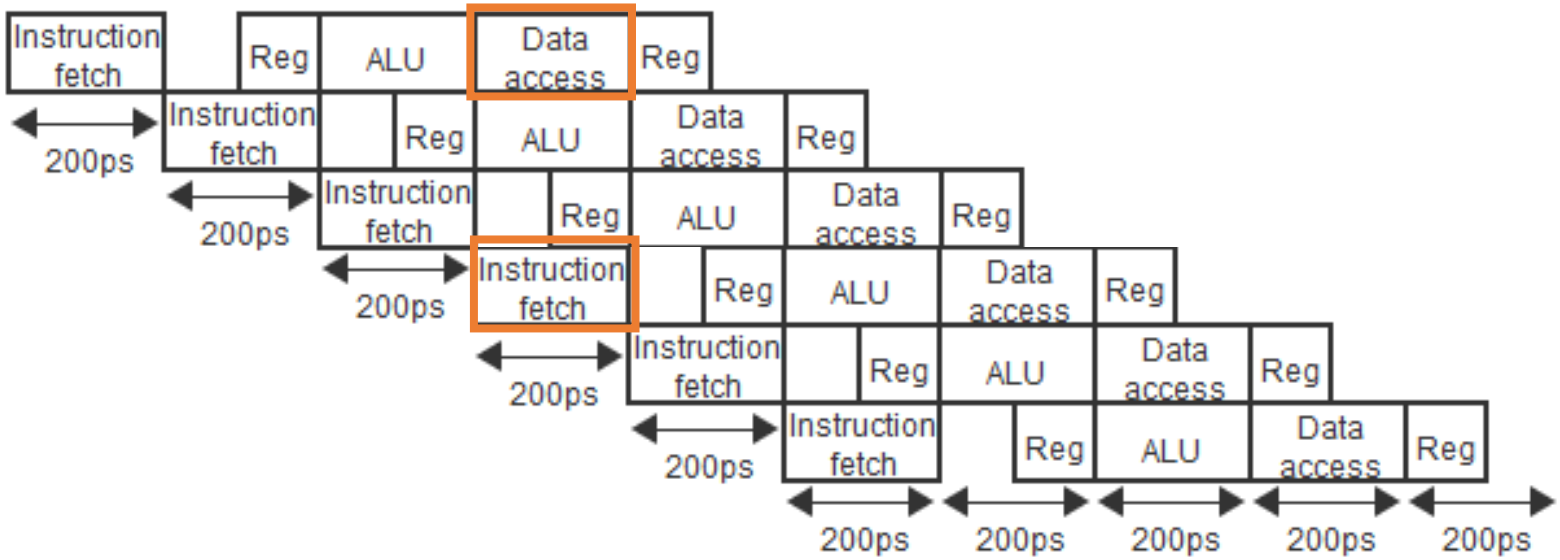
- Three types of hazards:
  - Structural hazard
  - Data hazard
  - Control hazard

Pipeline Hazards

- A **structural hazard** occurs when the hardware cannot support the combination of instructions that are set to execute.

- For example:
  - An add instruction is at the step where it should use the ALU
  - A branch instruction (parallel to the add instruction) is also at a step where it should use the ALU.
  - Both instructions cannot simultaneously use the ALU and such a situation is a structural hazard.

Pipeline Hazards



- If there weren't separate data and instruction memories, two instructions trying to access one memory component would result in a structural hazard

Pipeline Hazards

- A **data hazard** occurs when the pipeline must be stalled because data that is needed to execute the instruction is not yet available.

- For example:

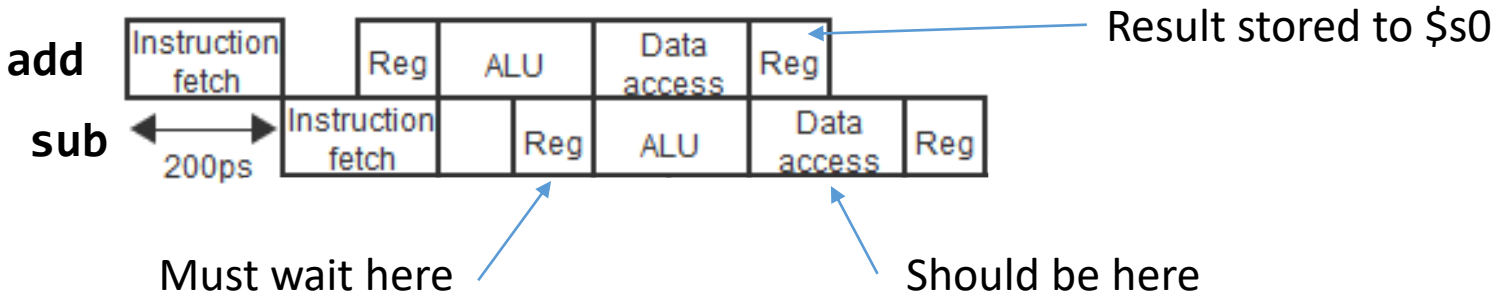
```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```

  - The second instruction relies on the previous instruction to finish.

Pipeline Hazards

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```

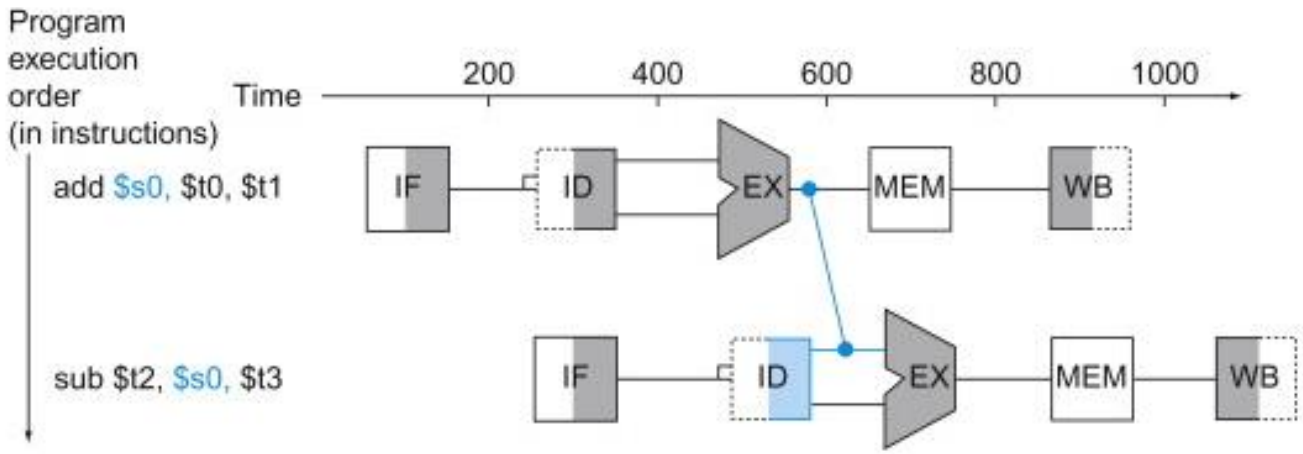
Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps



- Dependences like this happen often
  - Expecting compilers to prevent all data hazards is not realistic

Pipeline Hazards

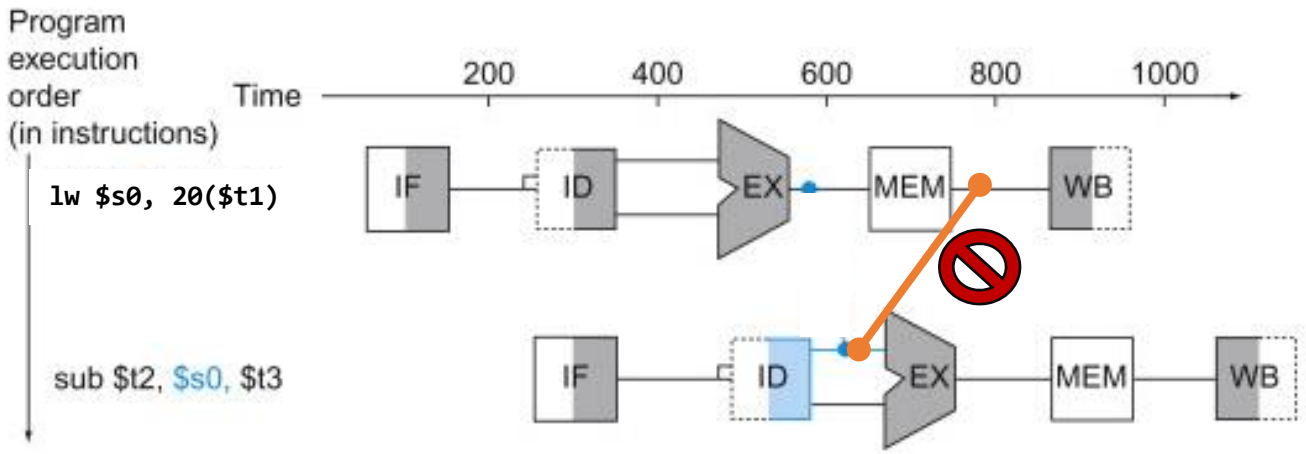
- **Forwarding** is a method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from registers or memory.



- If it were a load instruction to $s0 prior to the sub instruction (able to be forwarded after the MEM stage) then a stall would occur

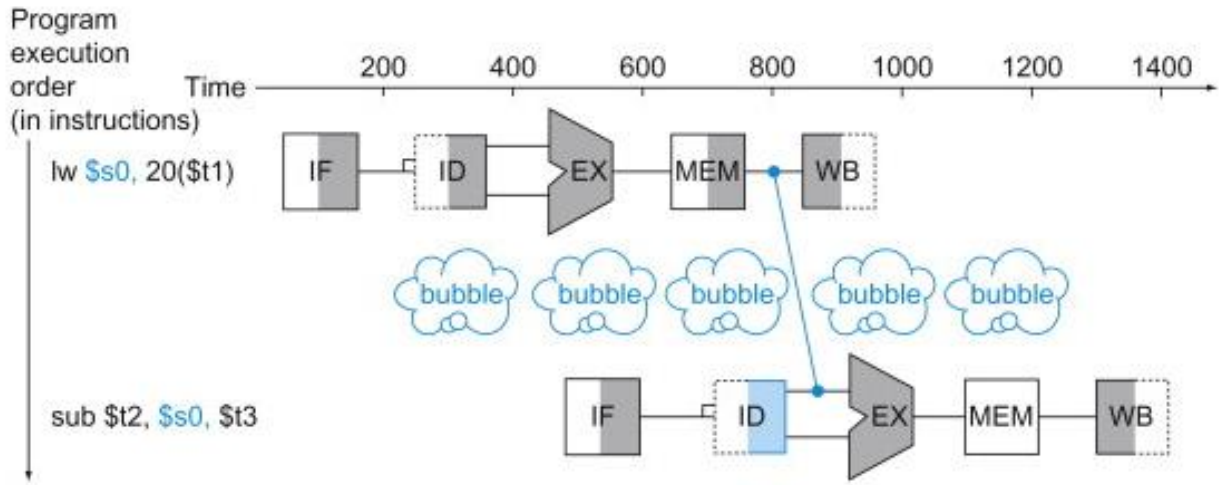
Pipeline Hazards

- If it were a load instruction to $s0 prior to the sub instruction (load can forward after the MEM stage) then a stall would still occur, even with forwarding.
  - Data cannot be forwarded backwards to when the sub instruction needed it
  - This is called a **load-use data hazard**, a specific type of data hazard in which the data being loaded by a load instruction has not yet become available when it is needed by another instruction



Pipeline Hazards

- A **pipeline stall** (or *bubble*) is a stall initiated to resolve a hazard



- Without the stall, the path from the memory access stage output to the execution stage input would be going backward in time, which is impossible.

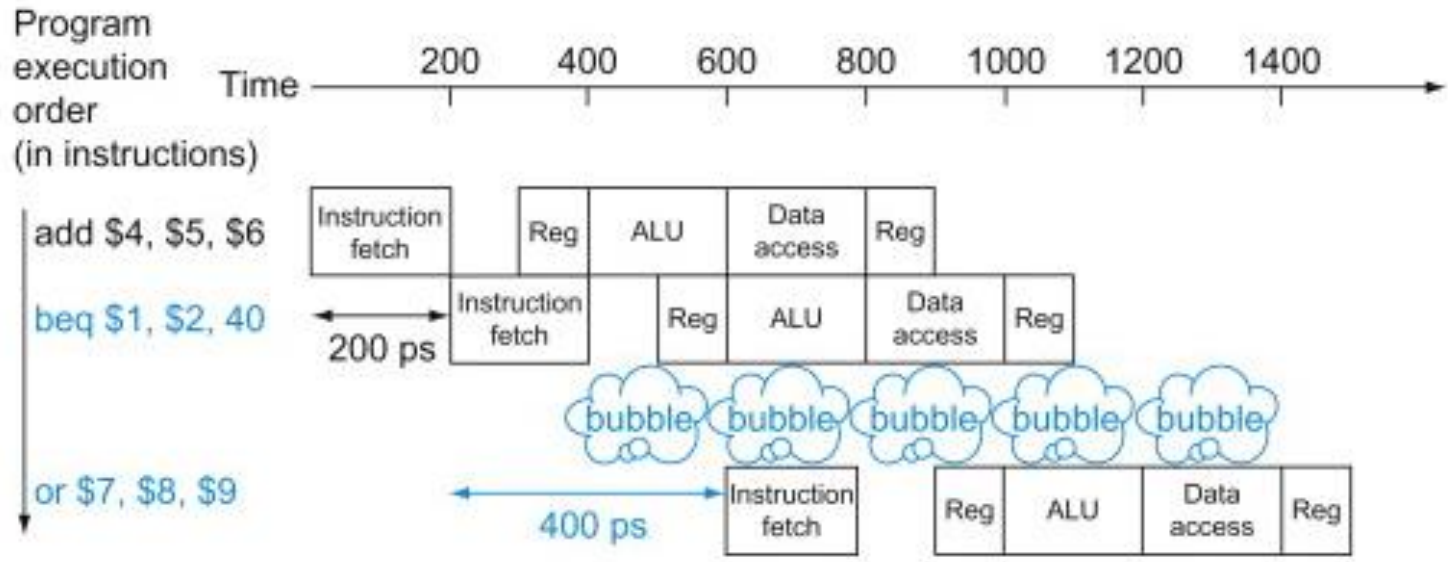
Pipeline Hazards

- A **control hazard** occurs when the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed.
  - The flow of instruction addresses is not what the pipeline expected

- Happens when branching
  - Following the branch, the next instruction must be fetched on the very next clock cycle.
    - The pipeline cannot know what the next instruction should be, since it only just received the branch instruction from memory.

Pipeline Hazards

- One solution is to stall on every conditional branch
  - However, this “stall on branch” solution leads to significant slow down



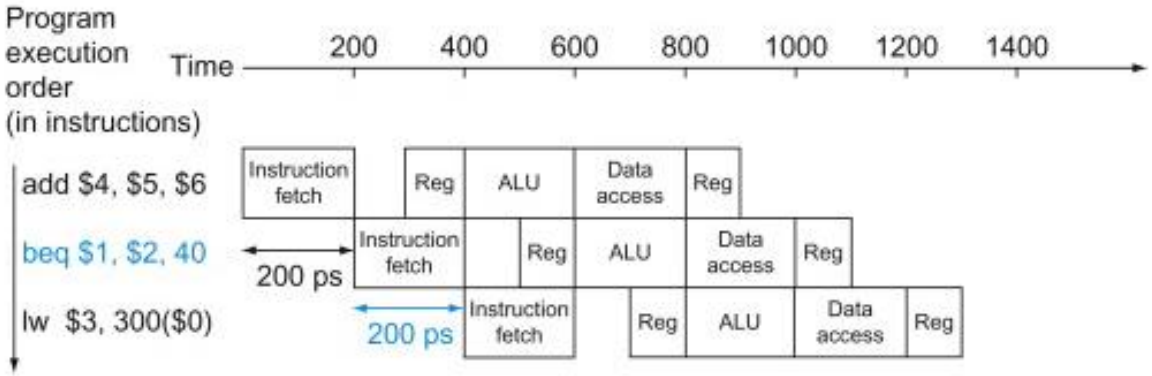
Pipeline Hazards

- A different approach is **prediction**

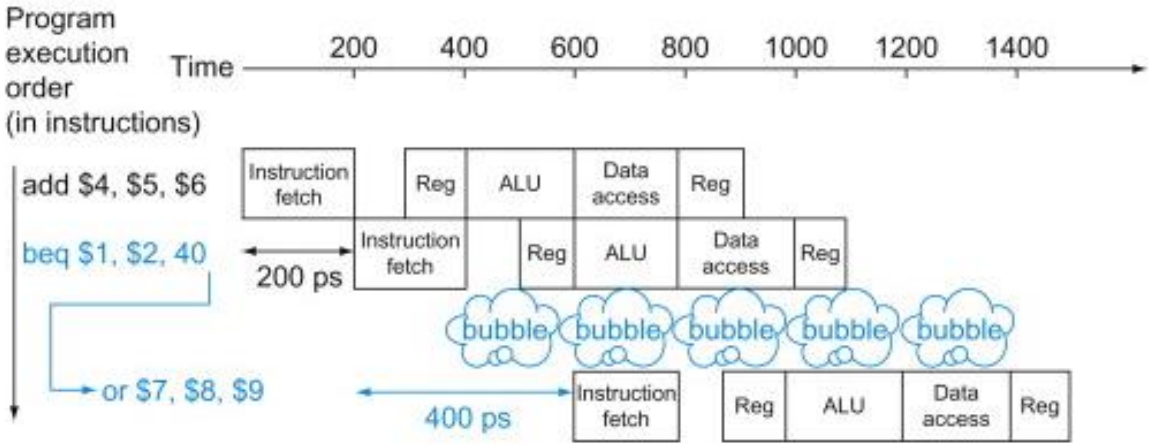
- One simple approach to prediction is assuming that branches will not be taken.
  - When correct, the pipeline proceeds without stalling.
  - Only when branches are taken will the pipeline stall.

Pipeline Hazards

Branch not taken;
No stall



Branch taken;
Stall



Pipeline Hazards

- A more sophisticated form of prediction is **branch prediction**

- Resolves a branch hazard by assuming a given outcome for the branch and proceeds from that assumption rather than waiting for the actual outcome.

- For example, at the bottom of loops are branches that jump back to the top of the loop.
  - They are likely to be taken and branch backward, so it’s reasonable to predict the branch will jump to an earlier address.

Pipeline Hazards

- **Dynamic branch prediction** makes a guess of which branch will be taken depending on the behavior of each branch
  - The prediction may change for a branch over the time the program is running.

- Keeps a history of each branch as taken or not taken
  - Then uses the recent past behavior to predict the future.

- If the guess is wrong, the pipeline control must ensure that the instructions following the wrongly guessed branch have no effect.
  - The pipeline must be restarted from the proper branch address.

- Branch predictors are a type of **speculative execution**.
  - The computer performs some work that might be needed; however, it the work might *not* be needed.

Interrupts

- **Interrupts** and **exceptions** are unscheduled events that disrupts program execution.
  - In many situations, both terms are used interchangeably
  - Not to be confused with a programming language's "exception", though the concept is similar

- *Interrupts* usually refer to an external event that changes normal flow of instruction execution, such as I/O device request

- Inefficient exception handling can reduce a processor's performance therefore the control unit must be designed to handle exceptions.

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

Interrupts

- An **imprecise interrupt** (or imprecise exception) are interrupts or exceptions in pipelined computers that are not associated with the exact instruction that was the cause of the interrupt or exception.

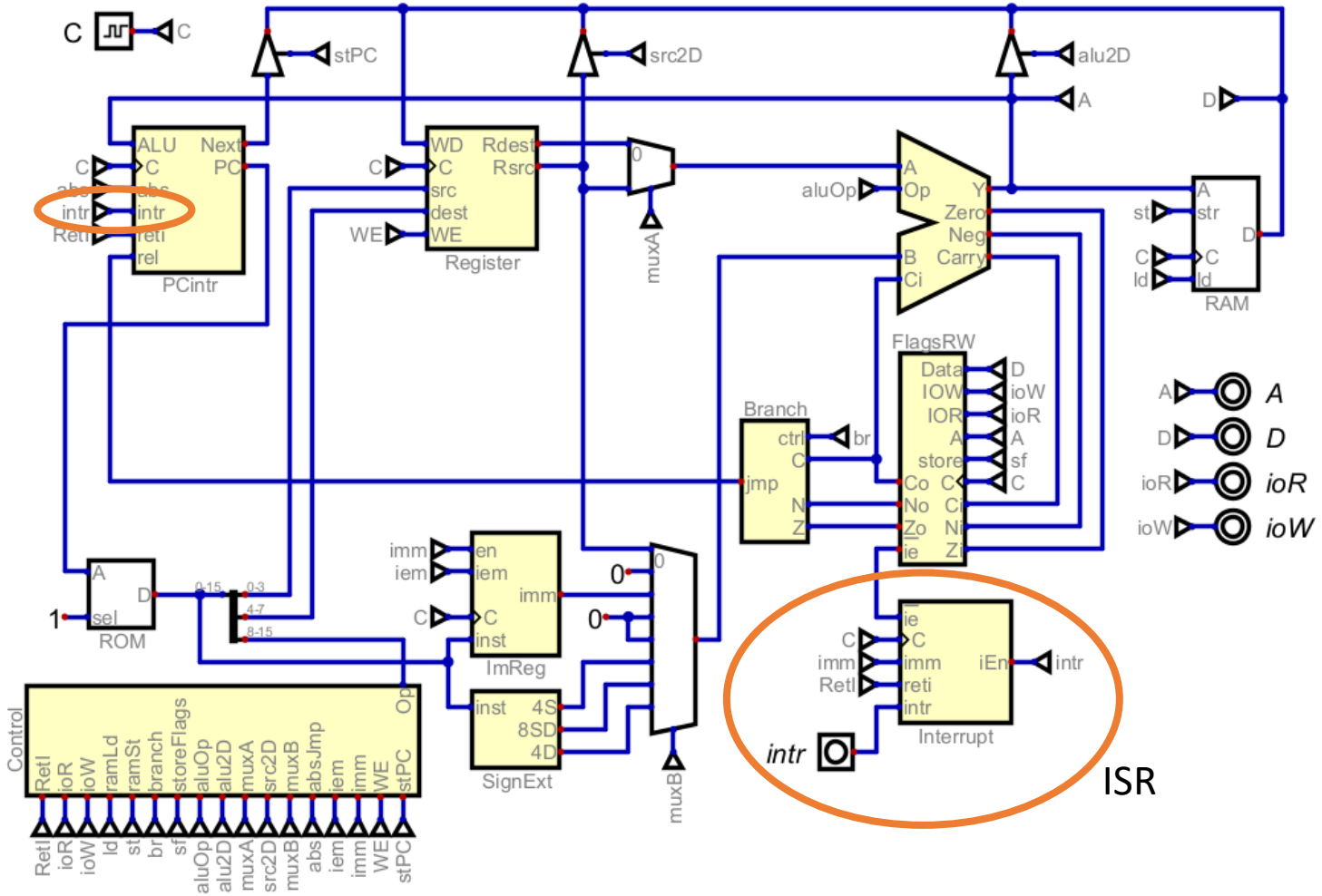
- A **precise interrupt** (or precise exception) are interrupts or exceptions that are associated with the correct instruction in pipelined computers.

Interrupts

- When an interrupt occurs, the processor finishes executing the currently instruction and transfers control to an **interrupt service routine** (ISR).
  - Also called an interrupt handler.

- Interrupts are usually caused by input devices
  - Hardware: Typing on the keyboard, moving the mouse, etc.
  - Software: Device drivers, software interrupts, system calls
    - (Like syscall in a MIPS simulator)

Interrupts


C
stPC
src2D
alu2D
A
D
PCintr
Register
muxA
aluOp
RAM
FlagsRW
Branch
A
D
ioR
ioW
ROM
ImReg
SignExt
muxB
Control
Interrupt
intr
ISR

Interrupts

- When an exception occurs, the processor saves the address of the instruction in the *exception program counter* (EPC).
  - Control is transferred to the operating system at some specified address.

- The operating system can take an appropriate action by
  - Providing some service to the user program
  - Taking some predefined action in response
  - Stopping the execution of the program and reporting an error.

Interrupts

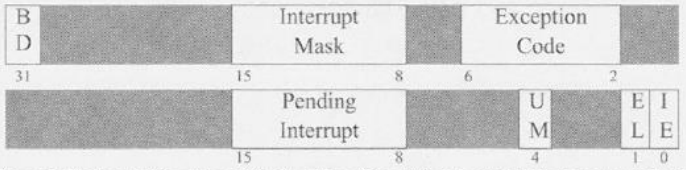
- After performing the action required because of the exception, the operating system can
  - Terminate the program
  - Continue its execution, using the EPC to determine where to restart the execution of the program.

- For the operating system to handle the exception, it must know the reason for the exception.
  - Also, the instruction that caused it.

Interrupts

- There are two primary methods used to communicate the reason for an exception.
  - One method used in the MIPS architecture is to include a status and cause register, which holds fields that indicates the reason for the exception.

EXCEPTION CONTROL REGISTERS: CAUSE AND STATUS



BD = Branch Delay, UM = User Mode, EL = Exception Level, IE =Interrupt Enable

EXCEPTION CODES

Number	Name	Cause of Exception	Number	Name	Cause of Exception
0	Int	Interrupt (hardware)	9	Bp	Breakpoint Exception
4	AdEL	Address Error Exception (load or instruction fetch)	10	RI	Reserved Instruction Exception
5	AdES	Address Error Exception (store)	11	CpU	Coprocessor Unimplemented
6	IBE	Bus Error on Instruction Fetch	12	Ov	Arithmetic Overflow Exception
7	DBE	Bus Error on Load or Store	13	Tr	Trap
8	Sys	Syscall Exception	15	FPE	Floating Point Exception

Interrupts

- A second method is to use *vectored interrupts*.
  - The address to which control is transferred is determined by the cause of the exception.
    - For example, to accommodate the two exception types above, we might define the two exception vector addresses.
  - The operating system knows the reason for the exception based on the address at which the exception is initiated.
    - When the exception is not vectored, one entry point for all exceptions can be used, and the operating system decodes the status register to find the cause.

Exception type	Exception vector address (in hex)
Undefined instruction	$8000\ 0000_{hex}$
Arithmetic overflow	$8000\ 0180_{hex}$

Interrupts

- (Part of the) x86 Interrupt Vector Table (IVT)

CPU Interrupt Layout

IVT Offset	INT #	Description
0x0000	0x00	Divide by 0
0x0004	0x01	Reserved
0x0008	0x02	NMI Interrupt
0x000C	0x03	Breakpoint (INT3)
0x0010	0x04	Overflow (INTO)
0x0014	0x05	Bounds range exceeded (BOUND)
0x0018	0x06	Invalid opcode (UD2)
0x001C	0x07	Device not available (WAIT/FWAIT)
0x0020	0x08	Double fault
0x0024	0x09	Coprocessor segment overrun
0x0028	0x0A	Invalid TSS
0x002C	0x0B	Segment not present
0x0030	0x0C	Stack-segment fault
0x0034	0x0D	General protection fault
0x0038	0x0E	Page fault
0x003C	0x0F	Reserved
0x0040	0x10	x87 FPU error
0x0044	0x11	Alignment check
0x0048	0x12	Machine check
0x004C	0x13	SIMD Floating-Point Exception
0x00xx	0x14-0x1F	Reserved
0x0xxx	0x20-0xFF	User definable