CSC 631: High-Performance Computer Architecture

Fall 2022
Lecture 3: Pipelining

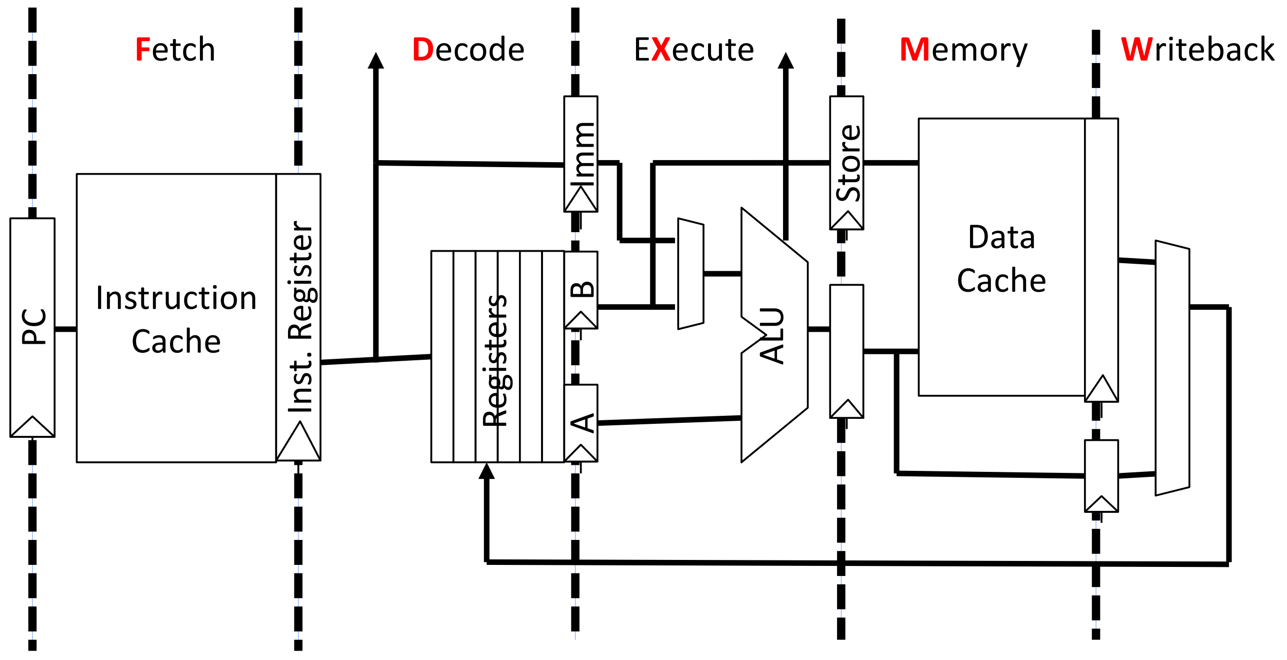
"Iron Law" of Processor Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

- Instructions per program depends on source code, compiler technology, and ISA
- Cycles per instructions (CPI) depends on ISA and µarchitecture
- Time per cycle depends upon the µarchitecture and base technology

Microarchitecture	CPI	cycle time
Microcoded	>1	short
Single-cycle unpipelined	1	long
Pipelined	1	short

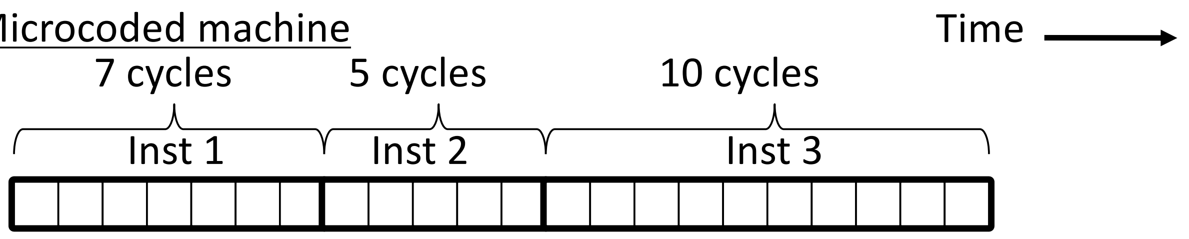
Classic 5-Stage RISC Pipeline



This version designed for regfiles/memories with synchronous reads and writes.

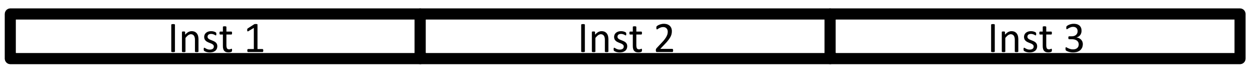
CPI Examples

Microcoded machine



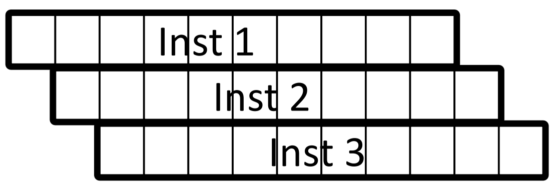
3 instructions, 22 cycles, CPI=7.33

Unpipelined machine



3 instructions, 3 cycles, CPI=1

Pipelined machine



3 instructions, 3 cycles, CPI=1

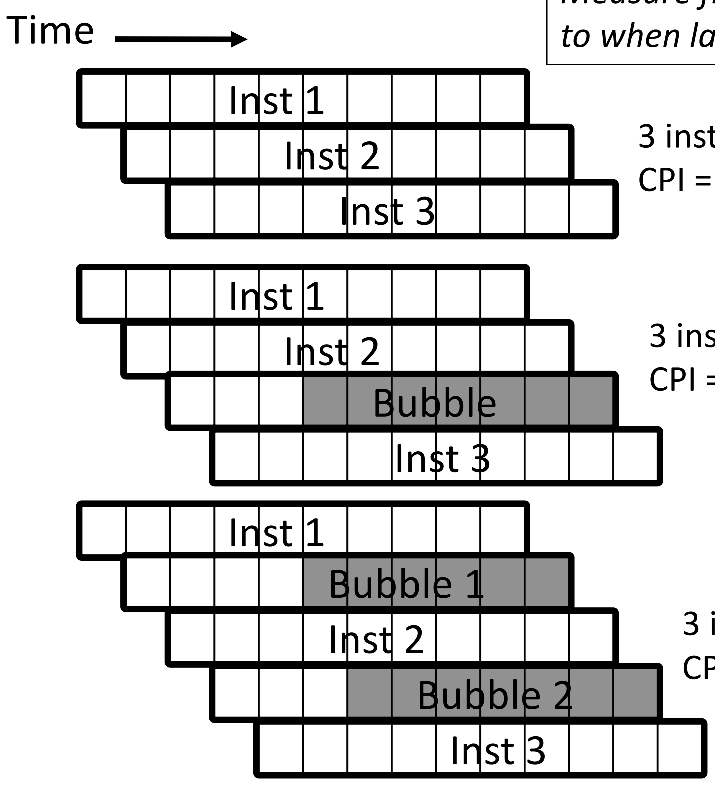
5-stage pipeline CPI≠5!!!

Instructions interact with each other in pipeline

- An instruction in the pipeline may need a resource being used by another instruction in the pipeline → *structural hazard*
- An instruction may depend on something produced by an earlier instruction
  - Dependence may be for a data value
    → *data hazard*
  - Dependence may be for the next instruction's address
    → *control hazard (branches, exceptions)*
- Handling hazards generally introduces bubbles into pipeline and reduces ideal CPI > 1

Pipeline CPI Examples

Measure from when first instruction finishes to when last instruction in sequence finishes.



3 instructions finish in 3 cycles
CPI = 3/3 =1

3 instructions finish in 4 cycles
CPI = 4/3 = 1.33

3 instructions finish in 5cycles
CPI = 5/3 = 1.67

Resolving Structural Hazards

- Structural hazard occurs when two instructions need same hardware resource at same time
  - Can resolve in hardware by stalling newer instruction till older instruction finished with resource
- A structural hazard can always be avoided by adding more hardware to design
  - E.g., if two instructions both need a port to memory at same time, could avoid hazard by adding second port to memory
- Classic RISC 5-stage integer pipeline has no structural hazards by design
  - Many RISC implementations have structural hazards on multi-cycle units such as multipliers, dividers, floating-point units, etc., and can have on register writeback ports

Types of Data Hazards

Consider executing a sequence of register-register instructions of type:

$$r_k \leftarrow r_i \text{ op } r_j$$

Data-dependence

$$\begin{tikzcd}[row sep=small] r_3 \arrow[drr] & \leftarrow & r_1 \text{ op } r_2 \\ r_5 & \leftarrow & r_3 \text{ op } r_4 \end{tikzcd}$$

Read-after-Write (RAW) hazard

Anti-dependence

$$\begin{tikzcd}[row sep=small] r_3 & \leftarrow & r_1 \text{ op } r_2 \arrow[dll] \\ r_1 & \leftarrow & r_4 \text{ op } r_5 \end{tikzcd}$$

Write-after-Read (WAR) hazard

Output-dependence

$$\begin{tikzcd}[row sep=small] r_3 \arrow[d, bend right] & \leftarrow & r_1 \text{ op } r_2 \\ r_3 & \leftarrow & r_6 \text{ op } r_7 \end{tikzcd}$$

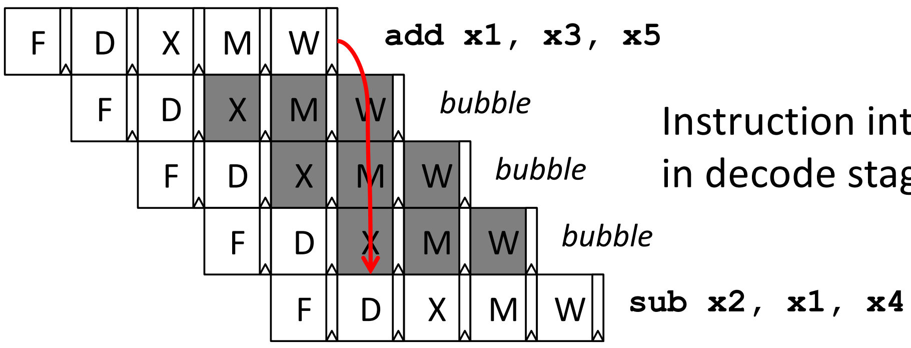
Write-after-Write (WAW) hazard

Three Strategies for Data Hazards

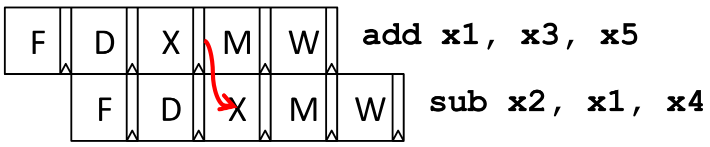
- Interlock
  - Wait for hazard to clear by holding dependent instruction in issue stage
- Bypass
  - Resolve hazard earlier by bypassing value as soon as available
- Speculate
  - Guess on value, correct if wrong

Interlocking Versus Bypassing

```
add x1, x3, x5
sub x2, x1, x4
```

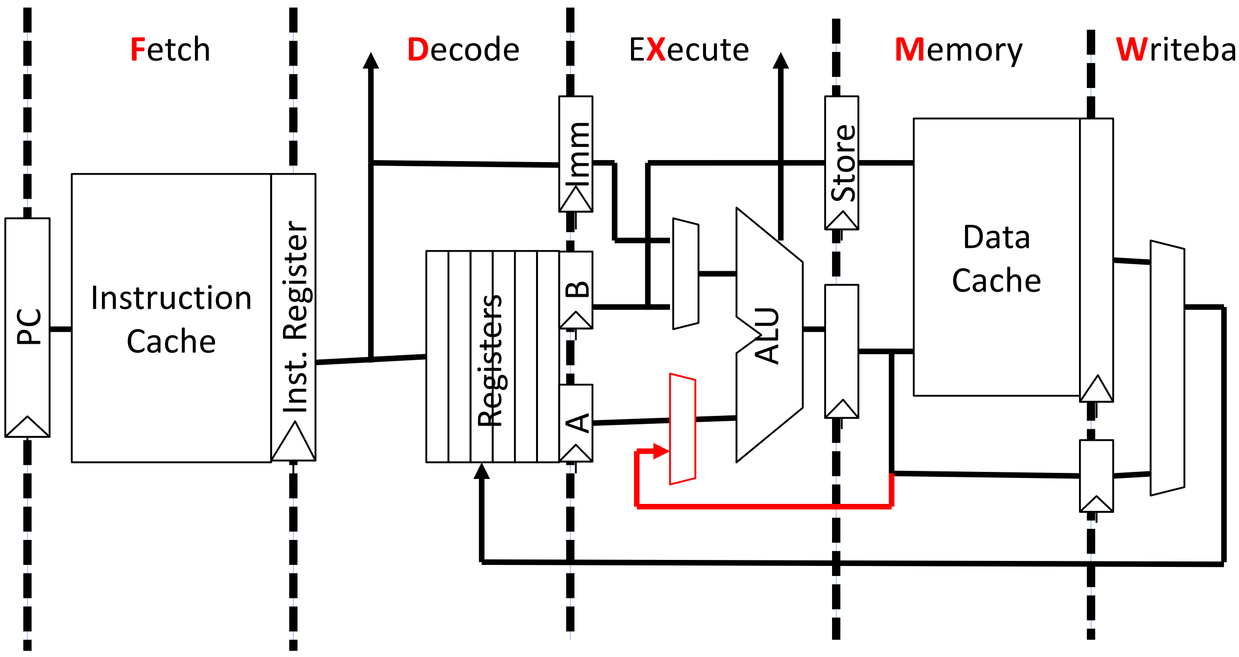


Instruction interlocked in decode stage

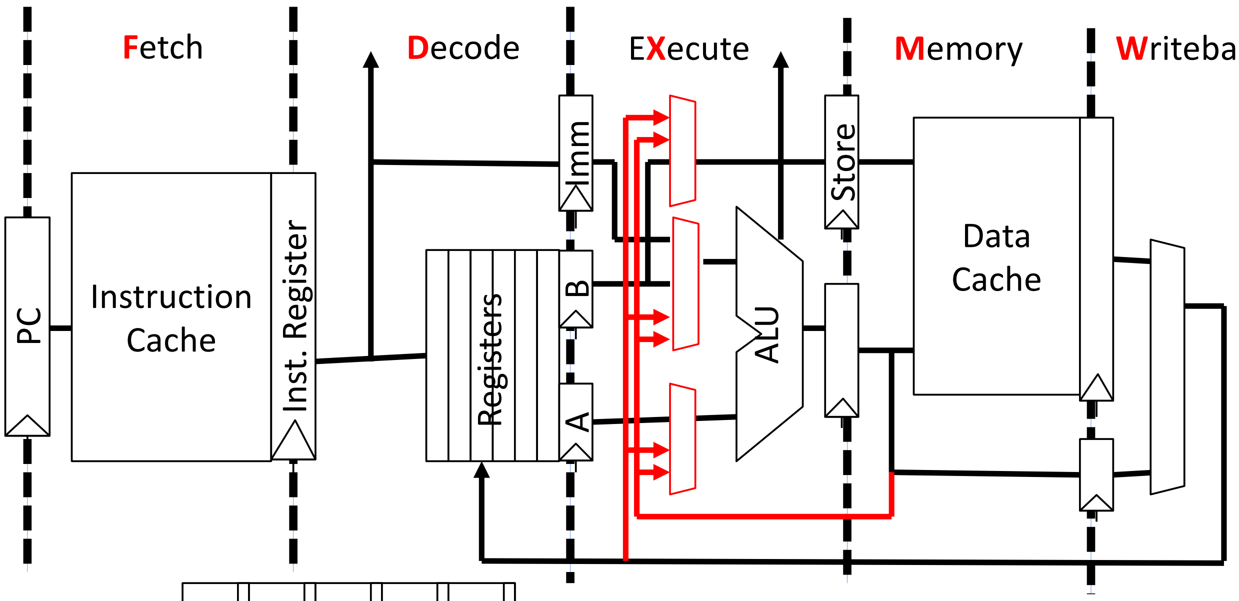


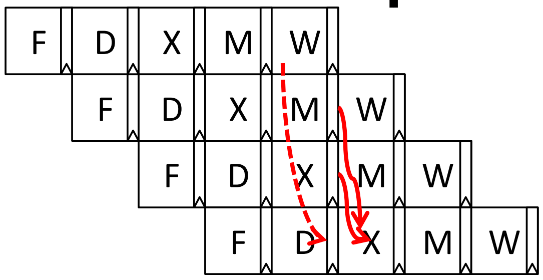
Bypass around ALU with no bubbles

Example Bypass Path



Fully Bypassed Data Path





[Assumes data written to registers in a W cycle is readable in parallel D cycle (dotted line). Extra write data register and bypass paths required if this is not possible.]

Value Speculation for RAW Data Hazards

- Rather than wait for value, can guess value!

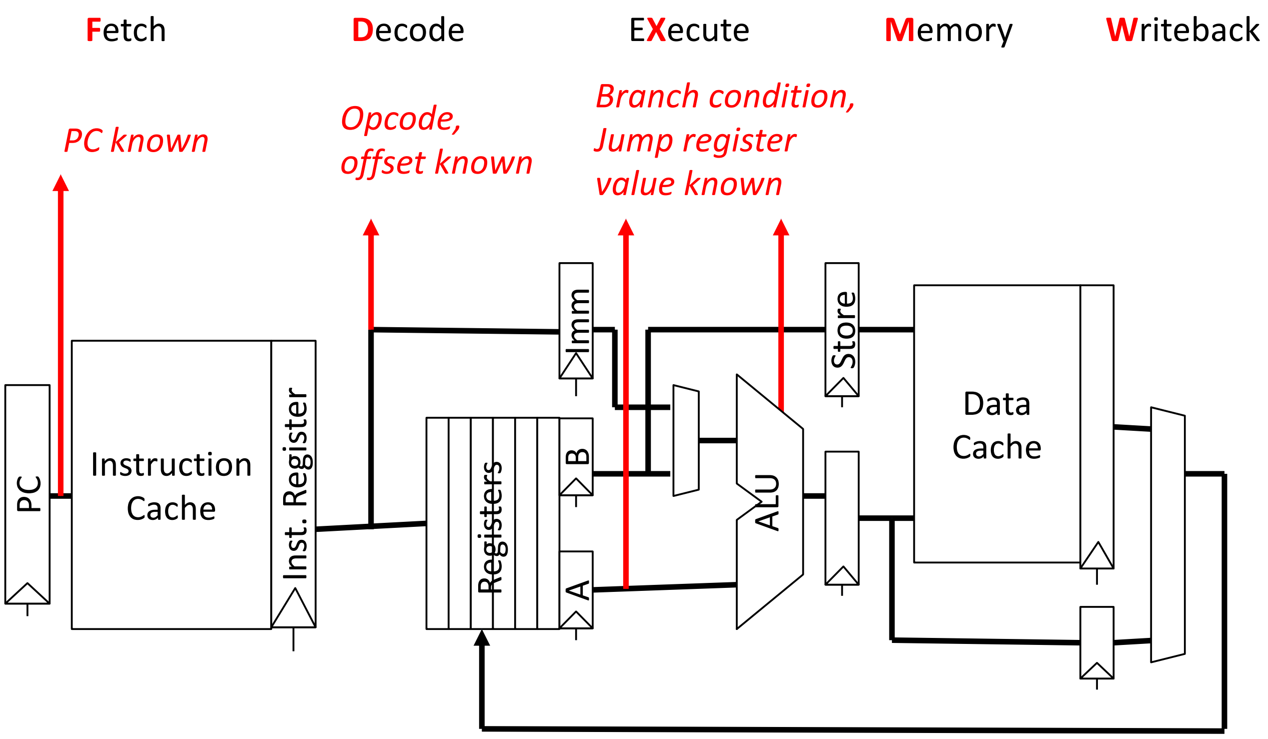
- So far, only effective in certain limited cases:
  - Branch prediction
  - Stack pointer updates
  - Memory address disambiguation

Control Hazards

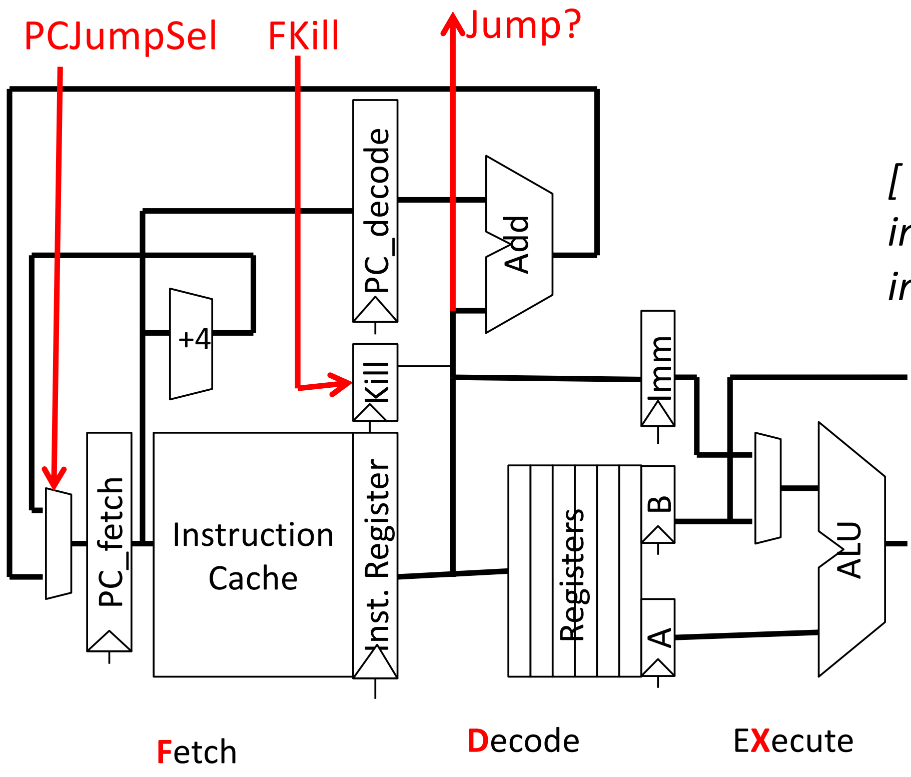
What do we need to calculate next PC?

- For Unconditional Jumps
  - Opcode, PC, and offset
- For Jump Register
  - Opcode, Register value, and offset
- For Conditional Branches
  - Opcode, Register (for condition), PC and offset
- For all other instructions
  - Opcode and PC (and have to know it’s not one of above)

Control flow information in pipeline

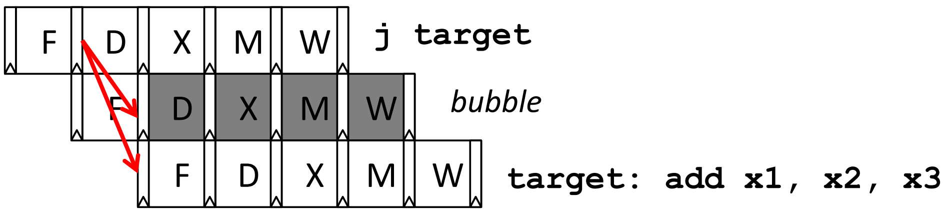


RISC-V Unconditional PC-Relative Jumps



[Kill bit turns instruction into a bubble]

Pipelining for Unconditional PC-Relative Jumps

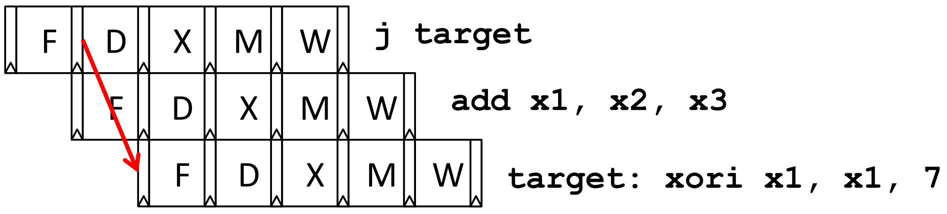


Branch Delay Slots

- Early RISCs adopted idea from pipelined microcode engines, and changed ISA semantics so instruction *after* branch/jump is always executed before control flow change occurs:

```
0x100 j target
0x104 add x1, x2, x3 // Executed before target
…
0x205 target: xori x1, x1, 7
```

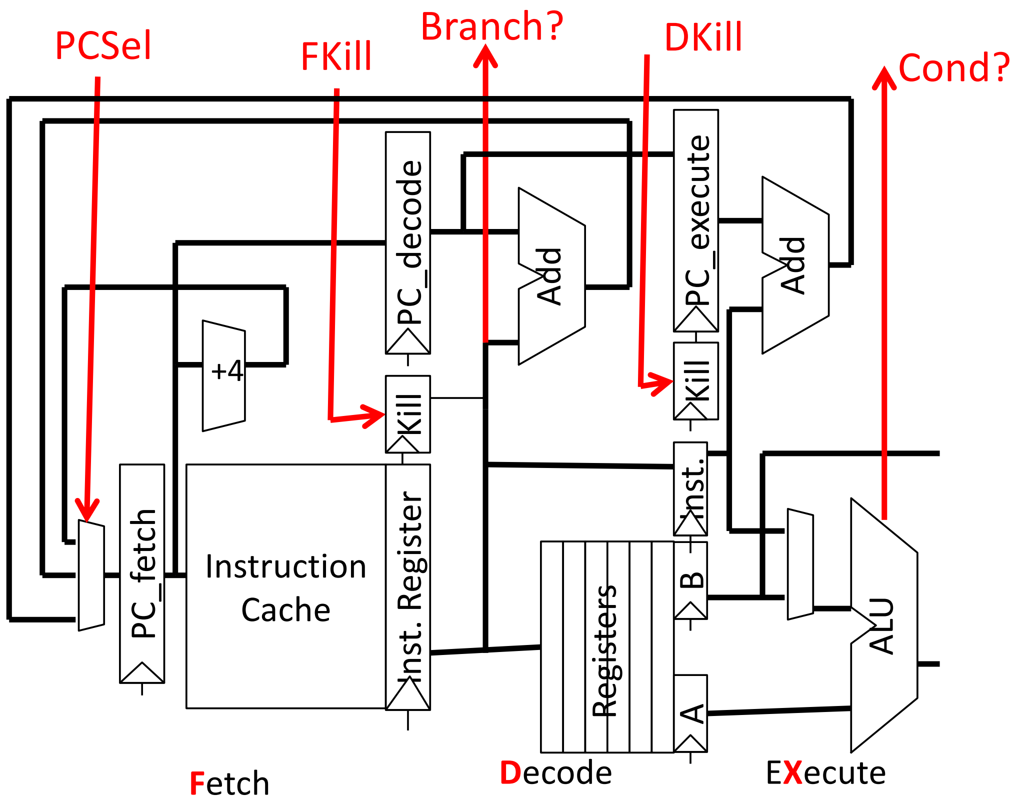
- Software has to fill delay slot with useful work, or fill with explicit NOP instruction



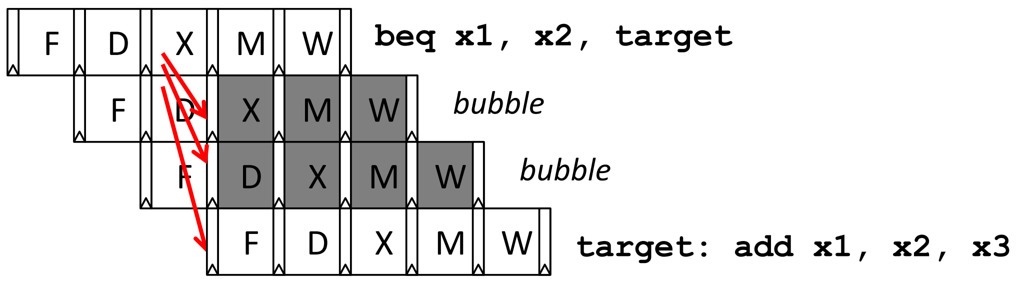
Post-1990 RISC ISAs don't have delay slots

- Encodes microarchitectural detail into ISA
  - c.f. IBM 650 drum layout
- Performance issues
  - Increased I-cache misses from NOPs in unused delay slots
  - I-cache miss on delay slot causes machine to wait, even if delay slot is a NOP
- Complicates more advanced microarchitectures
  - Consider 30-stage pipeline with four-instruction-per-cycle issue
- Better branch prediction reduced need
  - Will see branch prediction later on

RISC-V Conditional Branches

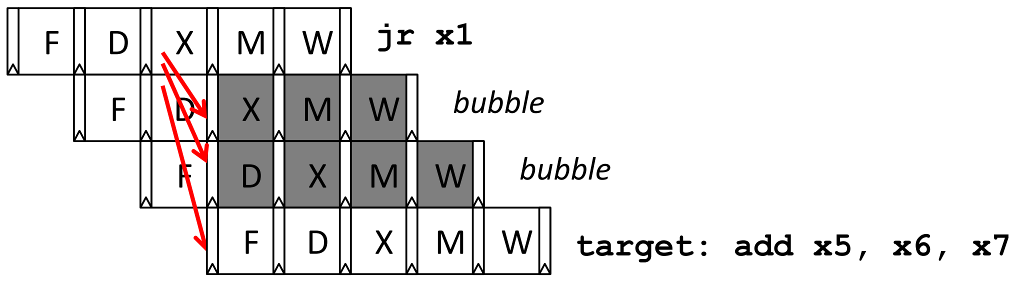


Pipelining for Conditional Branches



Pipelining for Jump Register

- Register value obtained in execute stage



Why instruction may not be dispatched every cycle in classic 5-stage pipeline (CPI>1)

- Full bypassing may be too expensive to implement
  - typically all frequently used paths are provided
  - some infrequently used bypass paths may increase cycle time and counteract the benefit of reducing CPI
- Loads have two-cycle latency
  - Instruction after load cannot use load result
  - MIPS-I ISA defined *load delay slots*, a software-visible pipeline hazard (compiler schedules independent instruction or inserts NOP to avoid hazard). Removed in MIPS-II (pipeline interlocks added in hardware)
    - MIPS:"**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages"
- Jumps/Conditional branches may cause bubbles
  - kill following instruction(s) if no delay slots

Machines with software-visible delay slots may execute significant number of NOP instructions inserted by the compiler.
NOPs reduce CPI, but increase instructions/program!

Traps and Interrupts

Recall the following definition from OS:

- ***Exception***: An unusual internal event caused by program during execution
  - E.g., page fault, arithmetic underflow
- ***Interrupt***: An external event outside of running program
- ***Trap***: Forced transfer of control to supervisor caused by exception or interrupt
  - Not all exceptions cause traps (c.f. IEEE 754 floating-point standard)

Asynchronous Interrupts

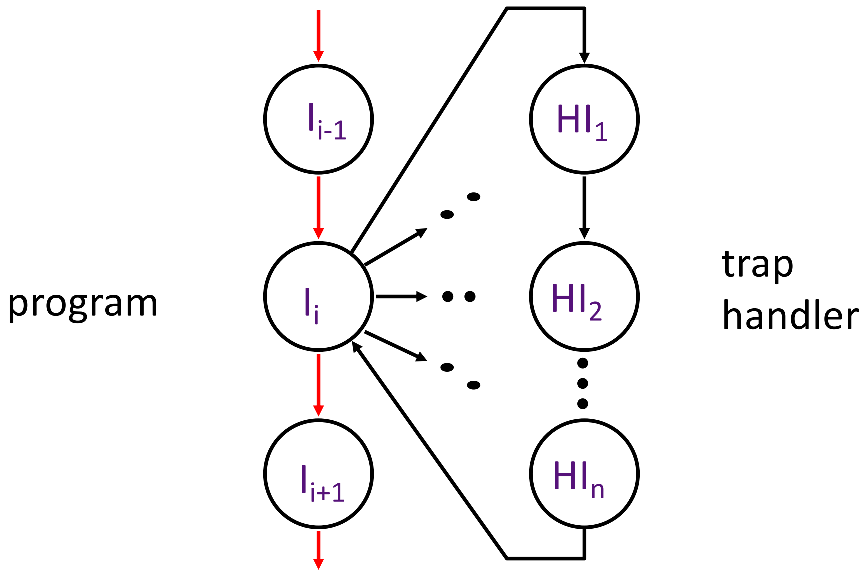
- An I/O device requests attention by asserting one of the *prioritized interrupt request lines*

- When the processor decides to process the interrupt
  - It stops the current program at instruction I_i , completing all the instructions up to I_{i-1} *(precise interrupt)*
  - It saves the PC of instruction I_i in a special register, **E**xception **P**rogram **C**ounter (EPC)
  - It disables interrupts and transfers control to a designated interrupt handler running in supervisor mode

Interrupt Handler

- Saves EPC before enabling interrupts to allow nested interrupts $\Rightarrow$
  - need an instruction to move EPC into GPRs
  - need a way to mask further interrupts at least until EPC can be saved
- Needs to read a *status register* that indicates the cause of the interrupt
- Uses a special indirect jump instruction ERET (*return-from-environment*) which
  - enables interrupts
  - restores the processor to the user mode
  - restores hardware status and control state

Trap: altering the normal flow of control



An *external or internal event* that needs to be processed by another (system) program. The event is usually unexpected or rare from program's point of view.

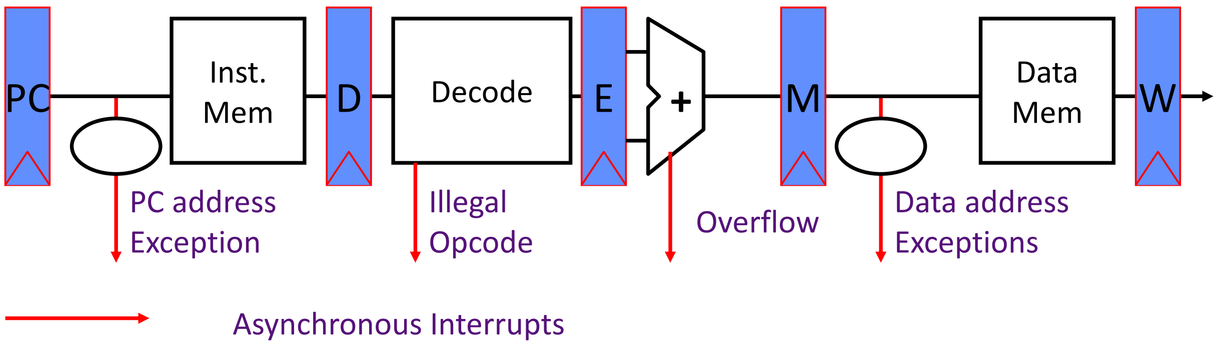
Trap Handler

- Saves ***EPC*** before enabling interrupts to allow nested interrupts ⇒
  - need an instruction to move EPC into GPRs
  - need a way to mask further interrupts at least until EPC can be saved
- Needs to read a *status register* that indicates the ***cause*** of the trap
- Uses a special indirect jump instruction ERET (*return-from-environment*) which
  - enables interrupts
  - restores the processor to the user mode
  - restores hardware status and control state

Synchronous Trap

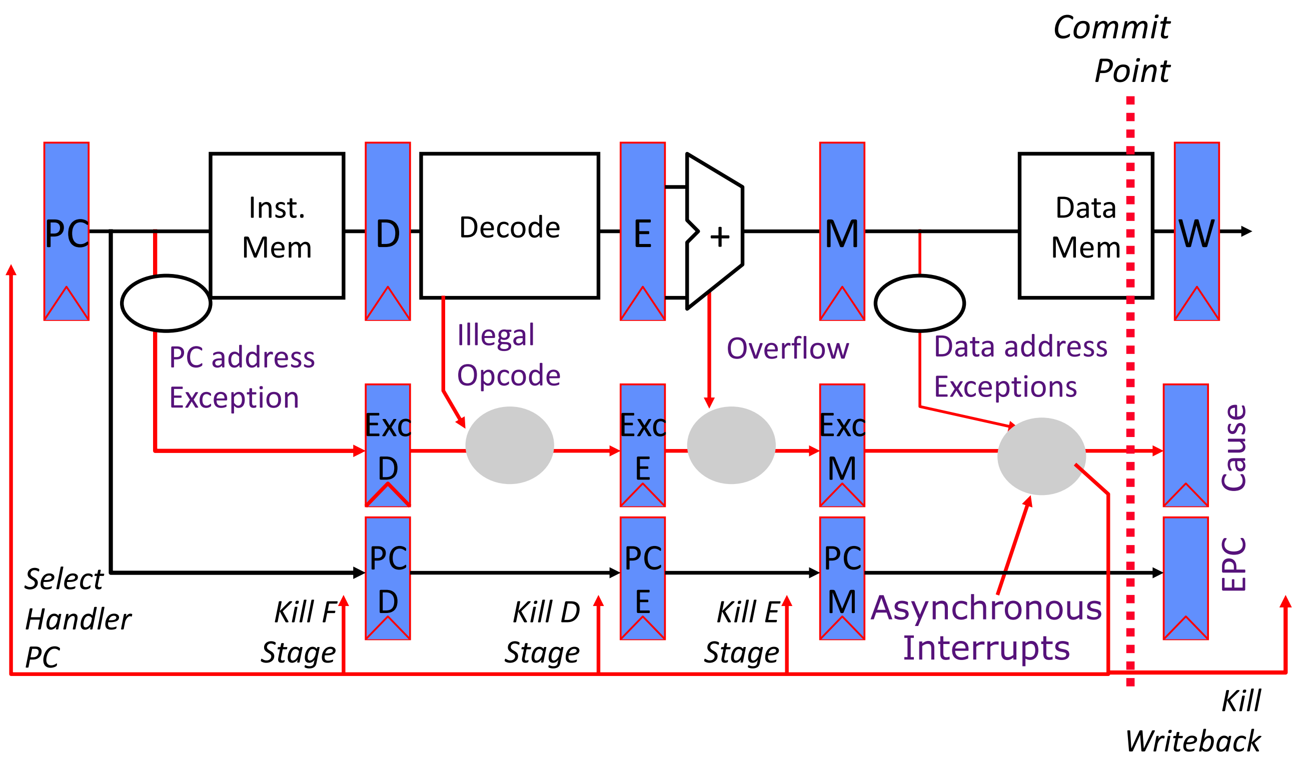
- A synchronous trap is caused by an exception on a *particular instruction*
- In general, the instruction cannot be completed and needs to be *restarted* after the exception has been handled
  - requires undoing the effect of one or more partially executed instructions
- In the case of a system call trap, the instruction is considered to have been completed
  - a special jump instruction involving a change to a privileged mode

Exception Handling 5-Stage Pipeline



- How to handle multiple simultaneous exceptions in different pipeline stages?
- How and where to handle external asynchronous interrupts?

Exception Handling 5-Stage Pipeline



Exception Handling 5-Stage Pipeline

- Hold exception flags in pipeline until commit point (M stage)
- Exceptions in earlier pipe stages override later exceptions *for a given instruction*
- Inject external interrupts at commit point (override others)
- If trap at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage

Speculating on Exceptions

- Prediction mechanism
  - Exceptions are rare, so simply predicting no exceptions is very accurate!
- Check prediction mechanism
  - Exceptions detected at end of instruction execution pipeline, special hardware for various exception types
- Recovery mechanism
  - Only write architectural state at commit point, so can throw away partially executed instructions after exception
  - Launch exception handler after flushing pipeline

- Bypassing allows use of uncommitted instruction results by following instructions

Deeper Pipelines: MIPS R4000

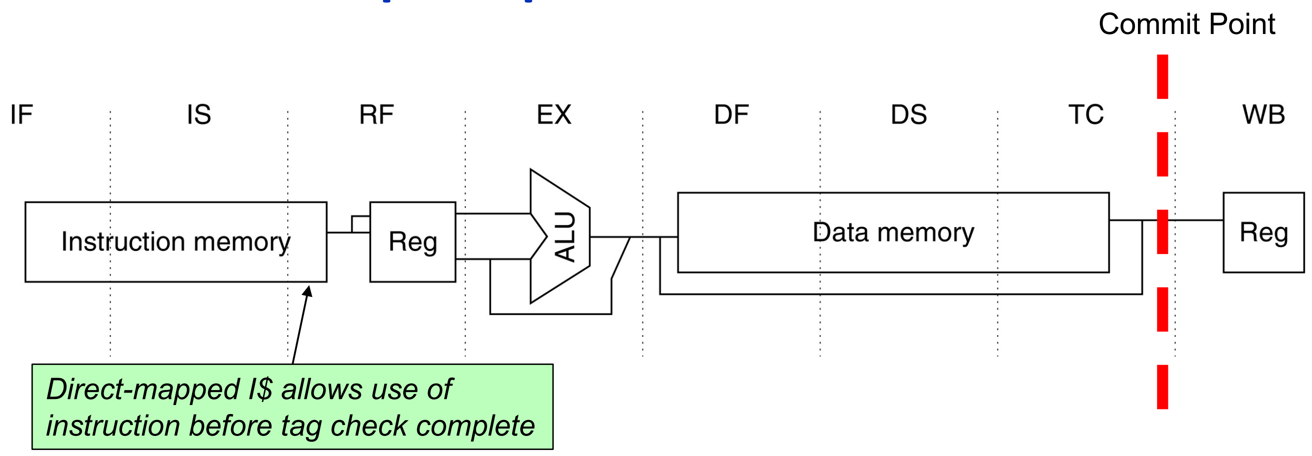

Commit Point
IF
IS
RF
EX
DF
DS
TC
WB
Instruction memory
Reg
ALU
Data memory
Reg
Direct-mapped I$ allows use of instruction before tag check complete

Figure C.36 The eight-stage pipeline structure of the R4000 uses pipelined instruction and data caches. The pipe stages are labeled. The vertical dashed lines represent the stage boundaries as well as the location of pipeline latches. The instruction is actually available at the end of IS, but the tag check is done in RF, while the registers are fetched. Thus, we show the instruction memory as operating through RF. The TC stage is needed for data memory access, because we cannot write the data into the register until we know whether the cache access was a hit or not.

R4000 Load-Use Delay

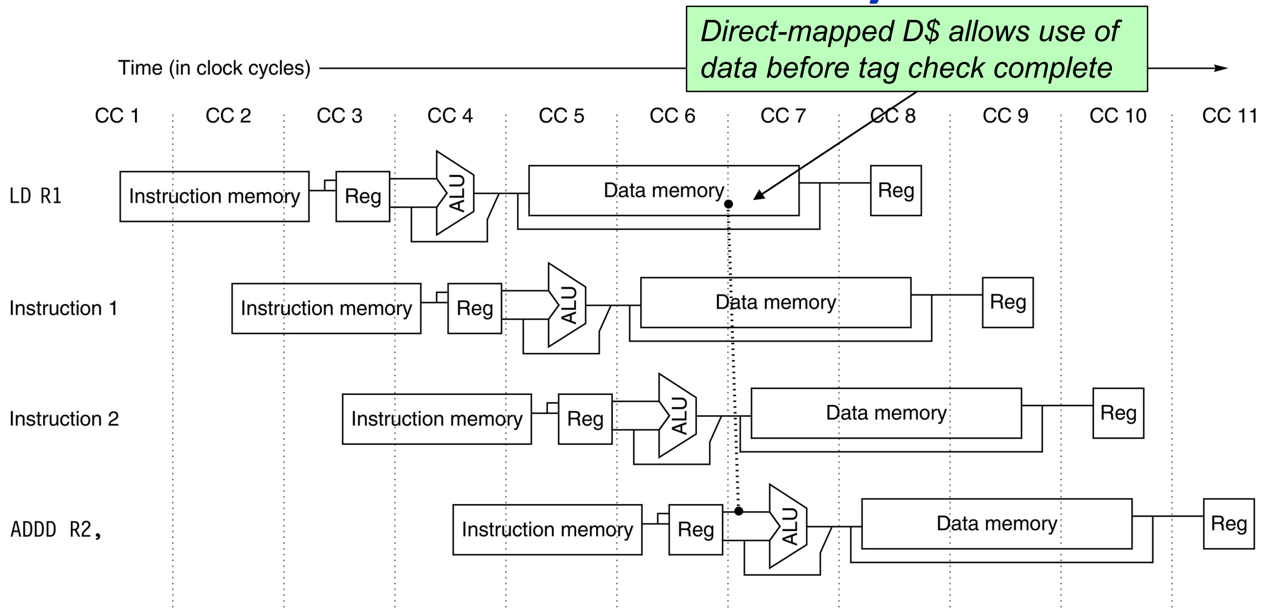


Figure C.37 The structure of the R4000 integer pipeline leads to a x1 load delay. A x1 delay is possible because the data value is available at the end of DS and can be bypassed. If the tag check in TC indicates a miss, the pipeline is backed up a cycle, when the correct data are available.

R4000 Branches

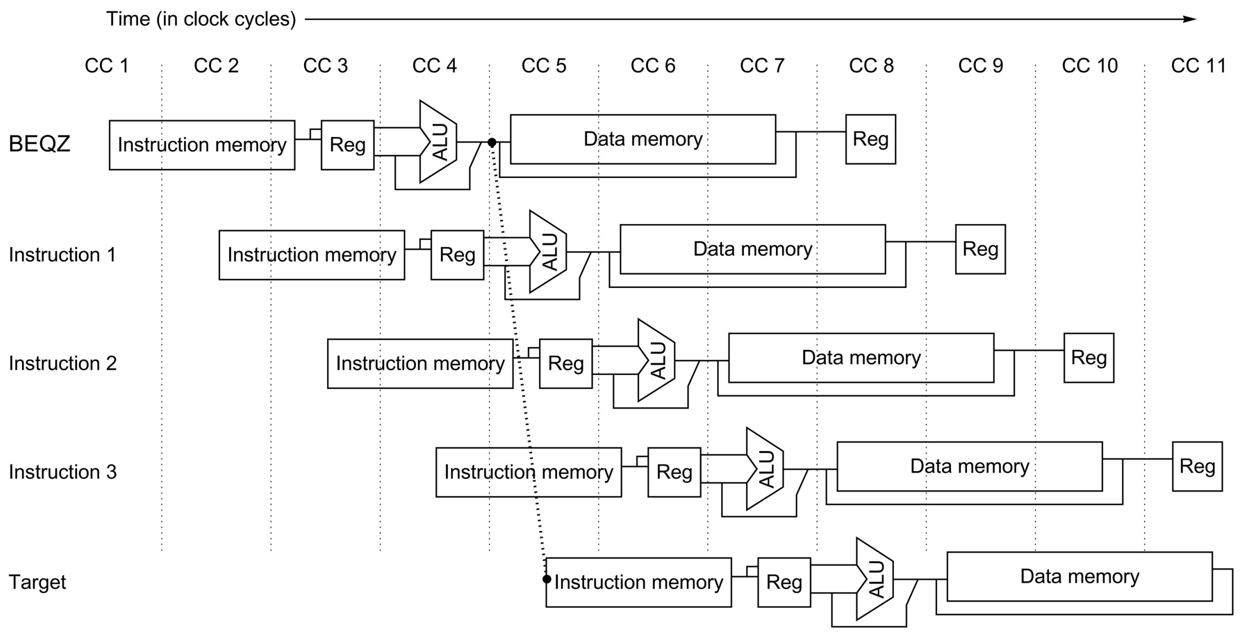


Figure C.39 The basic branch delay is three cycles, because the condition evaluation is performed during EX.

Supercomputers

Definitions of a supercomputer:

- Fastest machine in world at given task
- A device to turn a compute-bound problem into an I/O bound problem
- Any machine costing $30M+
- Any machine designed by Seymour Cray

- CDC6600 (Cray, 1964) regarded as first supercomputer

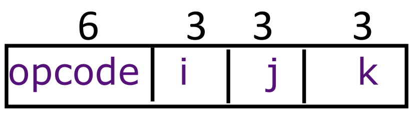
CDC 6600 ***Seymour Cray**, 1964*

- A fast pipelined machine with 60-bit words
  - 128 Kword main memory capacity, 32 banks
- Ten functional units (parallel, unpipelined)
  - Floating Point: adder, 2 multipliers, divider
  - Integer: adder, 2 incrementers, ...
- Hardwired control (no microcoding)
- *Scoreboard* for dynamic scheduling of instructions
- Ten Peripheral Processors for Input/Output
  - a fast multi-threaded 12-bit integer ALU
- Very fast clock, 10 MHz (FP add in 4 clocks)
- >400,000 transistors, 750 sq. ft., 5 tons, 150 kW, novel freon-based technology for cooling
- Fastest machine in world for 5 years (until 7600)
  - over 100 sold ($7-10M each)

CDC 6600: A Load/Store Architecture

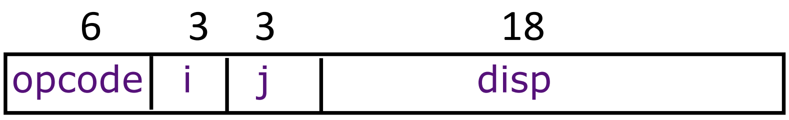
- Separate instructions to manipulate three types of reg.
  - 8x60-bit data registers (X)
  - 8x18-bit address registers (A)
  - 8x18-bit index registers (B)

- All arithmetic and logic instructions are register-to-register



$Ri \leftarrow Rj \text{ op } Rk$

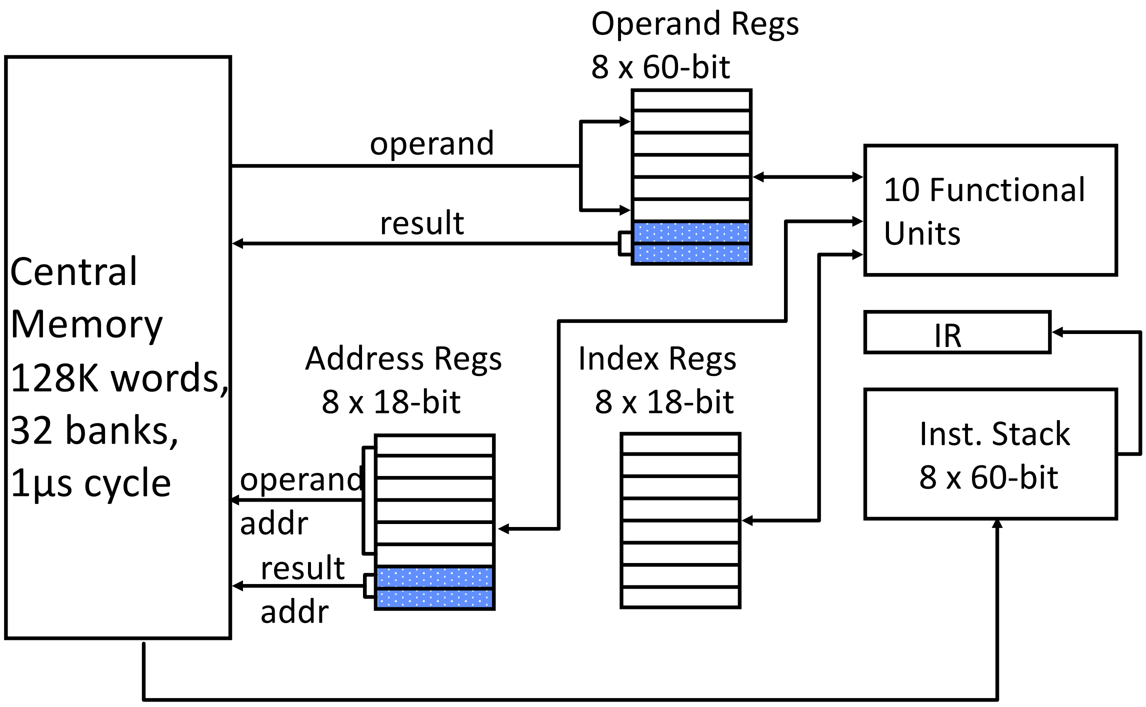
- Only Load and Store instructions refer to memory!



$Ri \leftarrow M[Rj + disp]$

Touching address registers 1 to 5 initiates a load
6 to 7 initiates a store
- very useful for vector operations

CDC 6600: Datapath



CDC6600: Vector Addition

```
       B0 ← - n
loop:  JZE  B0, exit
       A0 ← B0 + a0        load X0
       A1 ← B0 + b0        load X1
       X6 ← X0 + X1
       A6 ← B0 + c0        store X6
       B0 ← B0 + 1
       jump loop
```

Ai = address register
Bi = index register
Xi = data register

Computer Architecture Terminology

Latency (in seconds or cycles): Time taken for a single operation from start to finish (initiation to useable result)

Bandwidth (in operations/second or operations/cycle): Rate of which operations can be performed

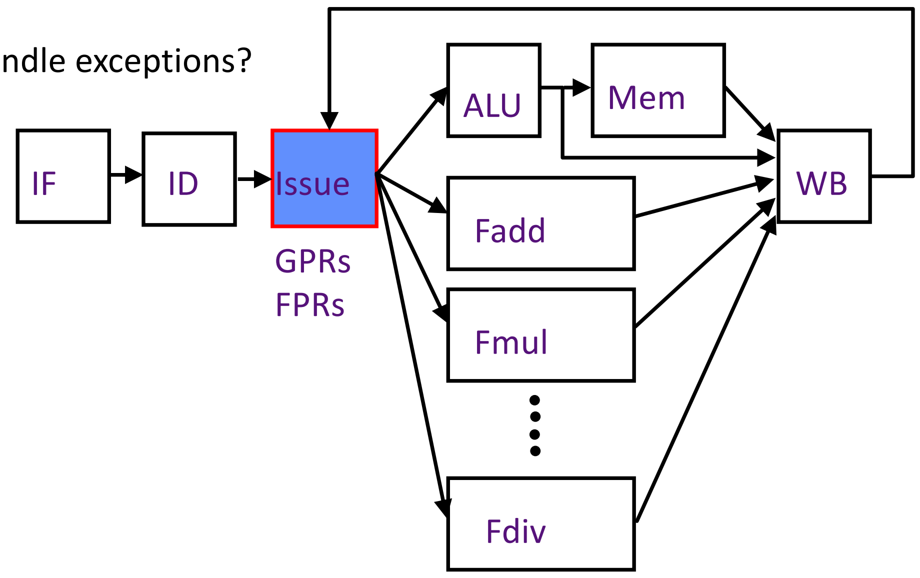
Occupancy (in seconds or cycles): Time during which the unit is blocked on an operation (structural hazard)

Note, for a single functional unit:

- Occupancy can be much less than latency (how?)
- Occupancy can be greater than latency (how?)
- Bandwidth can be greater than 1/latency (how?)
- Bandwidth can be less than 1/latency (how?)

Issues in Complex Pipeline Control

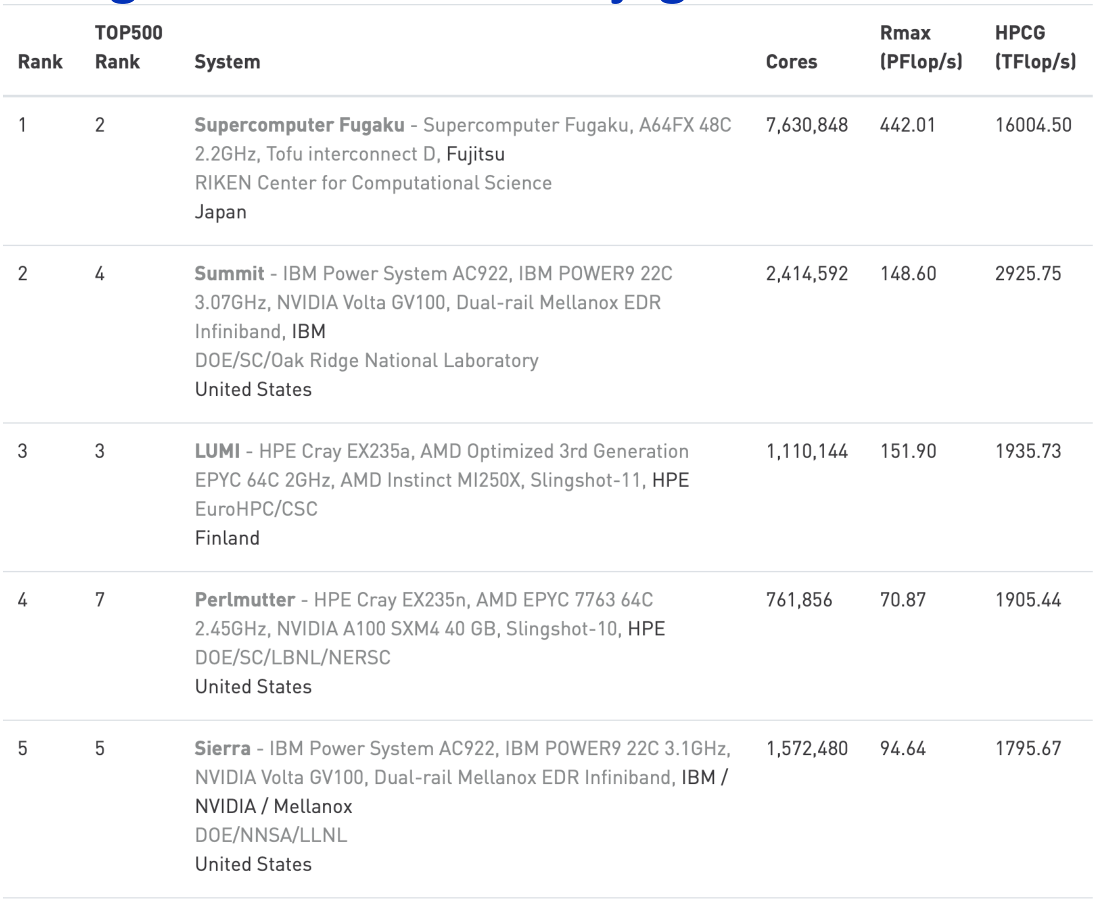
- Structural conflicts at the execution stage if some FPU or memory unit is not pipelined and takes more than one cycle
- Structural conflicts at the write-back stage due to variable latencies of different functional units
- Out-of-order write hazards due to variable latencies of different functional units
- How to handle exceptions?



Top500.org

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	1,110,144	151.90	214.35	2,942
4	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096
5	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94.64	125.71	7,438

High-Performance Conjugate Gradient



Rank	TOP500 Rank	System	Cores	Rmax (PFlop/s)	HPCG (TFlop/s)
1	2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	16004.50
2	4	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	2925.75
3	3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	1,110,144	151.90	1935.73
4	7	**Perlmutter** - HPE Cray EX235n, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10, HPE DOE/SC/LBNL/NERSC United States	761,856	70.87	1905.44
5	5	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94.64	1795.67

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
  - Krste Asanovic