ECSE 425 - Tutorial 3

RISC Architecture

Instruction Set Architecture (ISA)

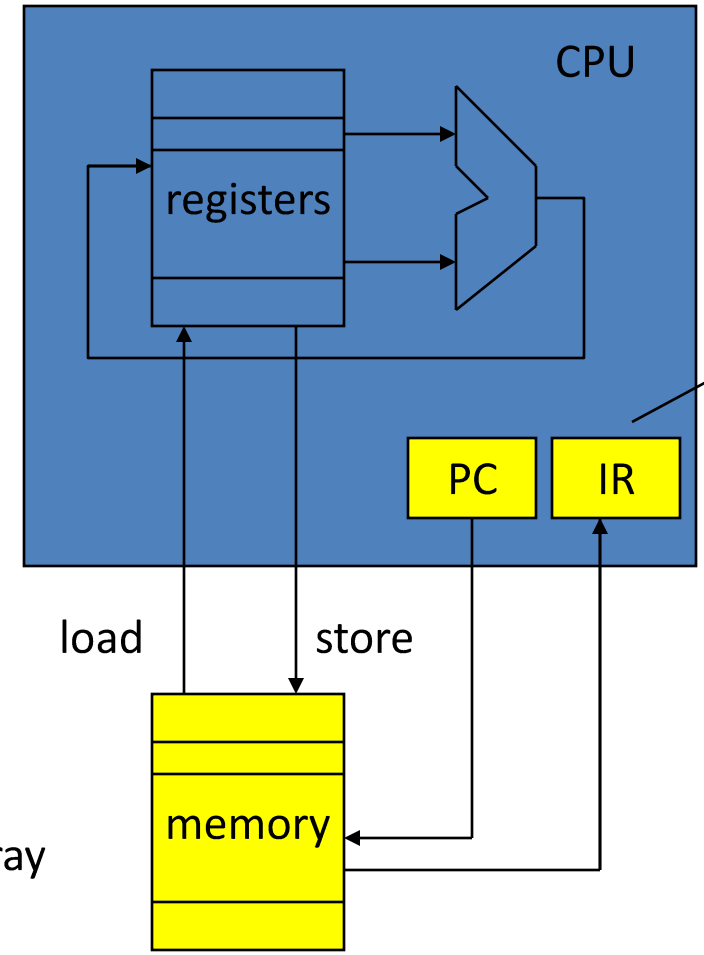
- Computers run programs made of simple operations called “instructions”
- The list of instructions offered by the machine is the “instruction set”
- The instruction set is what is visible to the programmer (really the compiler, although humans can directly program in “assembly language”).

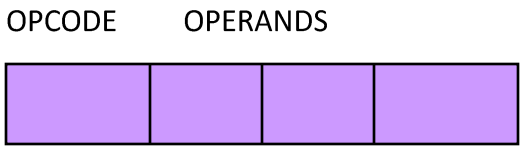
Instructions

- Two kinds of information in a computer:
  - instructions
  - data
- Instructions are stored as bits, just like data
- Instructions and data are stored in memory (or memories)

Basic Computer Organization

Limited number of **fast** registers for temporary storage

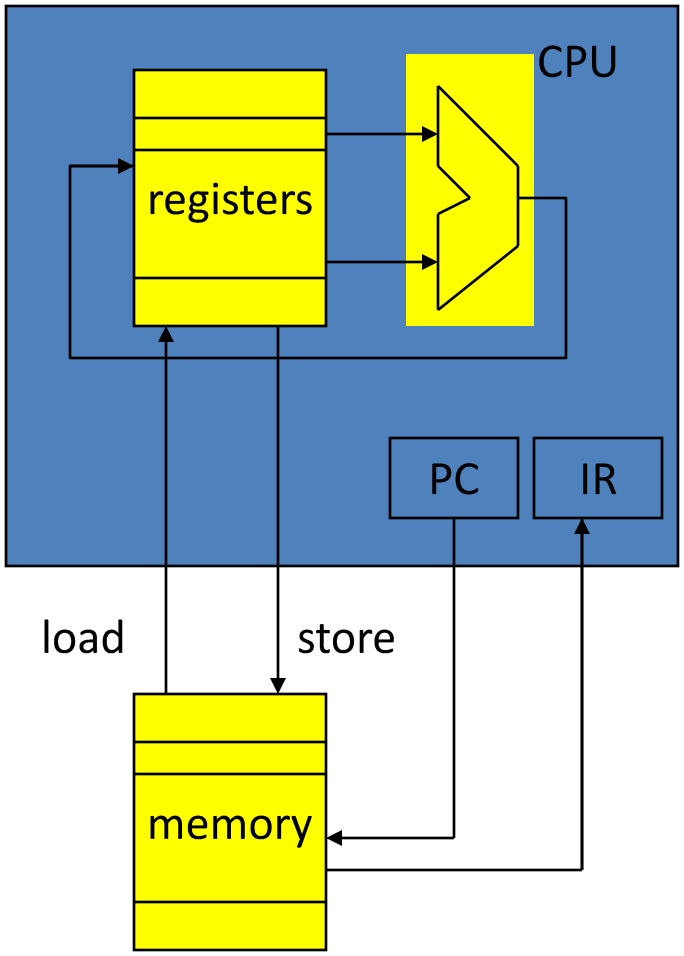




Instructions are loaded into an Instruction register (IR) from the address pointed to by the program counter (PC). The PC is incremented by the instruction size (in bytes) for each new instruction.
E.g. PC ← PC + 4

Large amount of **slow** memory
Arranged as an array of bytes

Load/Store Architecture (Reg-Reg)

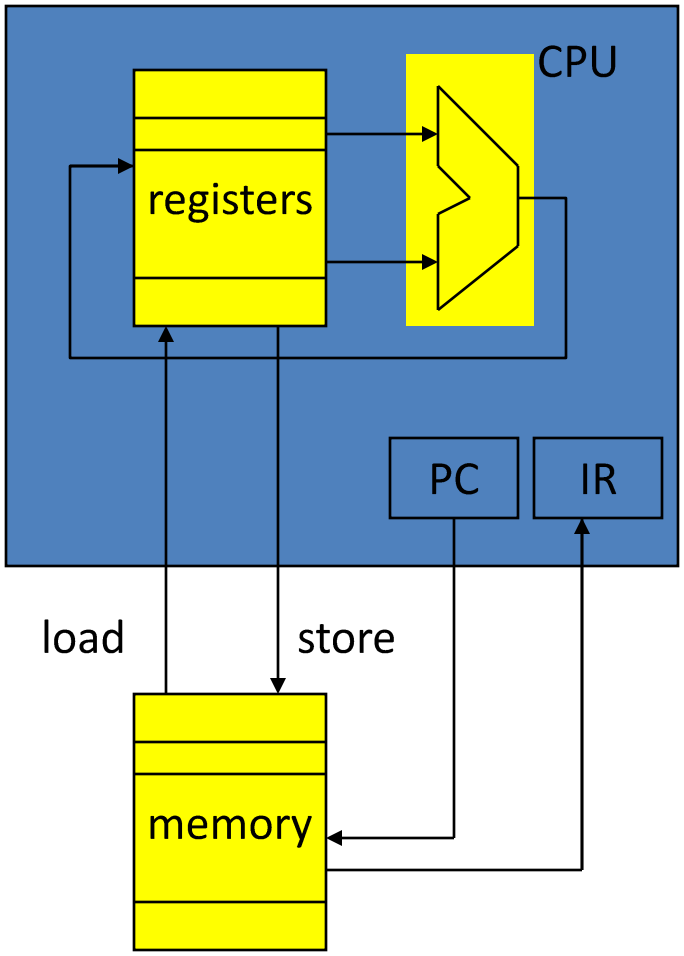


- Instructions can **ONLY** get their data and write their result from/to registers.

- The register numbers are specified in the operand fields of the instruction

- Since data is stored in memory, we need special "load" and "store" instructions for transfers between registers and memory. These two instructions are the ONLY ones allowed to access memory

Load/Store Architecture (Reg-Reg)



• **RISC** architectures are load/store. The regularity of this architecture enables fast organizations using **pipelining**.

• **CISC** machines (e.g. Intel IA-32) permit instructions to get their data from both registers and memory (mem-reg). These highly irregular architectures (mem-reg, variable-length instructions) are practically impossible to pipeline.

  • The advantage of them is that they produce shorter programs (no loads or stores needed, variable-length instr.), but memory today is cheap and compilers can't really use complex instructions anyways.

• Modern "CISC" machines really just translate the CISC instructions to a set of RISC instructions and run those.

  • Done purely for compatibility reasons.

MIPS64 - Architecture

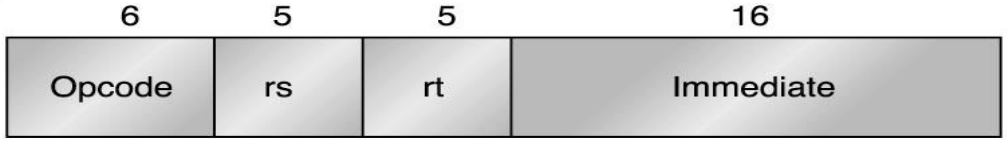
- RISC architecture => Load-Store
- 32-bit instructions
- 31 * 64-bit GPRs
  - R1, …, R31
  - R0 is hardwired to zero (and writing to it does nothing)
- 32 FPR
  - F0, …, F31

MIPS64 - Data Types

- Integer
  - 8-bits (byte)
  - 16-bits (short or half-word)
  - 32-bits (word)
  - 64-bits (double word)
- Floating point
  - 32-bits (single-precision)
  - 64-bits (double-precision)

MIPS64 - Instruction Format

I-type instruction



Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)

Conditional branch instructions (rs is register, rd unused)
Jump register, jump and link register
(rd = 0, rs = destination, immediate = 0)

R-type instruction

Register-register ALU operations: rd ← rs funct rt
Function encodes the data path operation: Add, Sub, . . .
Read/write special registers and moves

J-type instruction

Jump and jump and link
Trap and return from exception

Instruction Types

Instruction types:

- ALU instructions
  - +, -, /, *, %, &, |, ^, >>, <<, >, <, ==, etc…
- Load/store
  - Get a value/store a value in memory
- Branches and jumps
  - Modify the Program Counter (PC): PC_{next} != PC+4

Addressing Modes

- Immediate (constants)

```
ADD R4, #3          Regs[R4] ← Regs[R4] + 3
```

- Register-Register

```
ADD R4, R5, R6      Regs[R4] ← Regs[R5] + Regs[R6]
```

- Displacement (computed addresses, pointers, local variables, array accesses)

```
LD R4, 100(R3)      Regs[R4] ← Mem[Regs[R3] + 100]
```

- For others (not really used in RISC too often), see Figure B.6.

RISC Instructions

Instruction types:

- ALU instructions: reg-reg or reg-imm
  - reg←ALU(reg,reg) : **ADD R3,R1,R2**
  - reg←ALU(reg,imm) : **SUB R3, R1, #2**
- Data transfers (load/store): reg-imm
  - reg←MEM(ALU(reg,imm)) : **LOAD R3, 4(R1)**
  - MEM(ALU(reg,imm))←reg : **STORE R3, 0(R1)**
- Control (Branches and jumps): reg-imm
  - PC ← PC + imm if cond(reg,reg) : **BEQ R3, R4, label**

Instruction Types

Arithmetic and Logical	Add, subtract, and, or, shifts, multiply, divide.
Data Transfer	Load, Store
Control	Branch, jump, procedure call, return, trap.
System	Syscall, Virtual memory management
Floating Point	FPadd, FPmult, FPdiv, FPcompare
Decimal	Arithmetic and conversion
Strings	Move, copy, compare, search
Graphics	Pixel, Vertex ops, compress, decompress

MIPS64 – Sample Program

(5 * 6) + (7 * 8) (C = high–level language)

```
DADDUI   R1, R0, #5
DADDUI   R2, R0, #6
DMULU    R3, R1, R2
DADDUI   R1, R0, #7
DADDUI   R2, R0, #8
DMULU    R4, R1, R2
DADDU    R5, R3, R4
```

See Figure B.26 for a subset of the MIPS64 instruction set

Instruction Types

- It is often the case that few instruction statistically dominate.
  - e.g. SPEC92 benchmark indicates (80x86):

Loads:	**22%**
Branches:	**20%**
Compare:	**16%**
Store:	**12%**
ALU:	**19%**

- Important conclusions:
  - 5 (simple) types make 89% of all instructions
    - make these fast!
  - twice as many loads than stores (more reads than writes)

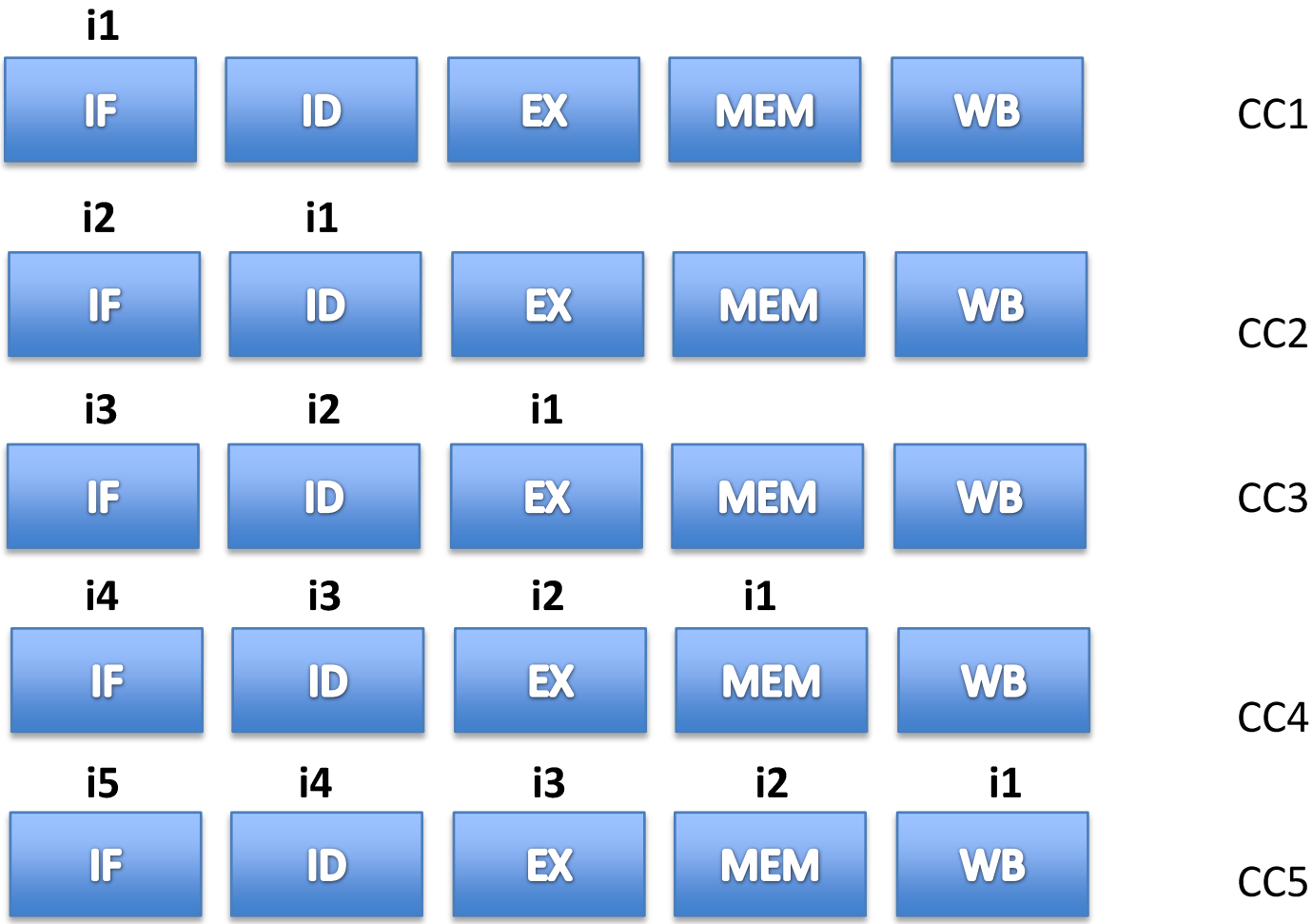
Pipelining

Classic 5-stage RISC pipeline:

- IF : fetch instruction [memory, read]
- ID : decode [register file, read]
  - Read from register file, sign-extend imm, comparisons
- EX : execute [ALU]
  - Used for: eff. mem. addr., reg-reg and reg-imm.
- MEM : access memory [memory, r/w]
- WB : write-back in registers [register file, write]

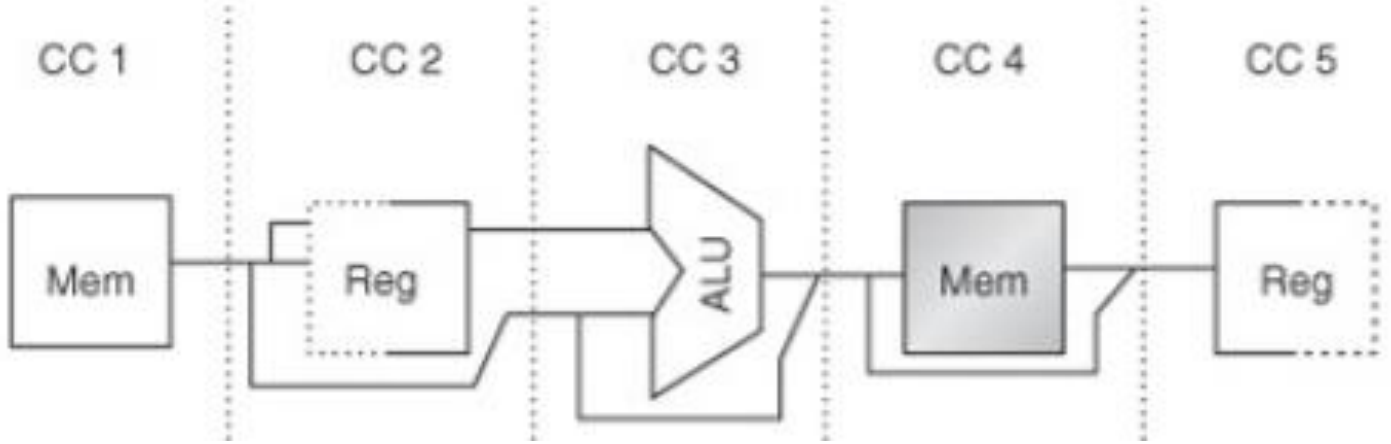
Pipelining

Classic 5-stage RISC pipeline:



Pipelining

Classic 5-stage RISC pipeline:

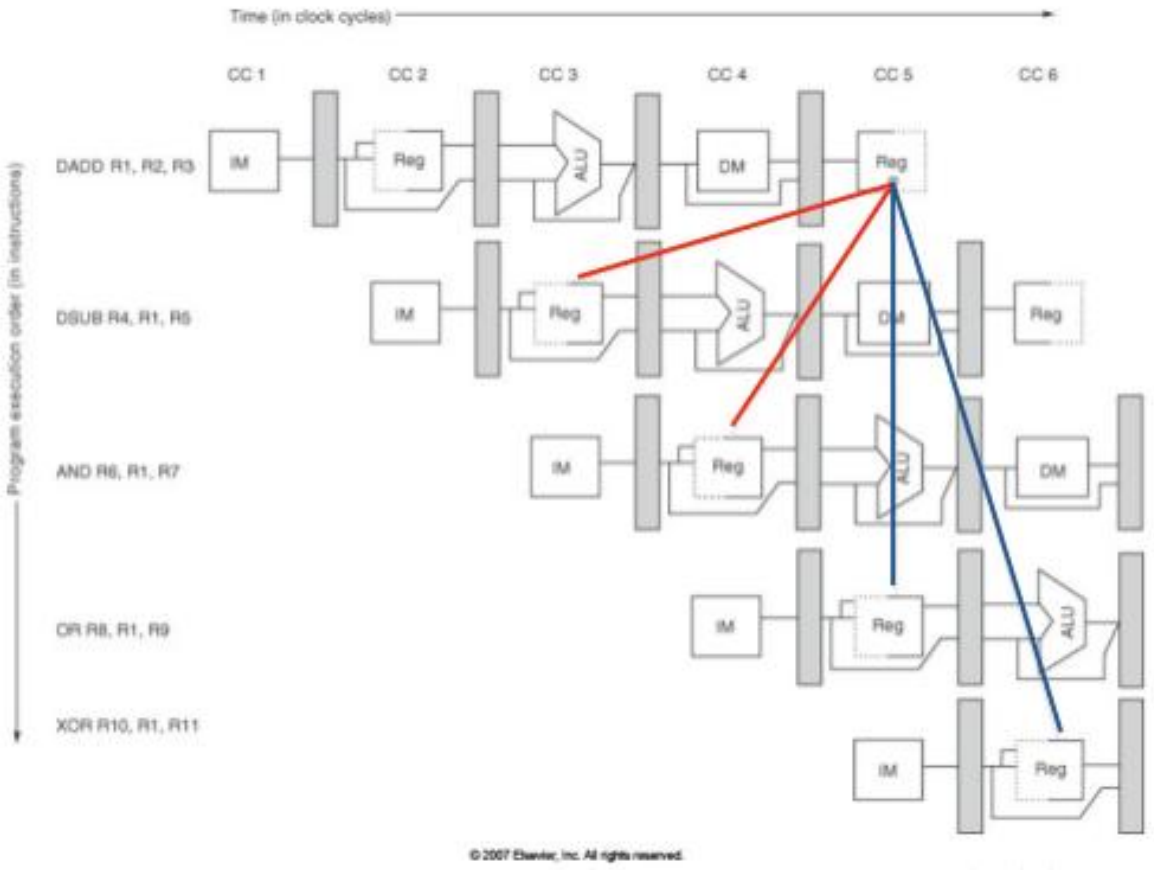


Hazards

- Structural hazards: multiple stages can't run at the same time because they share a resource
- Data hazards: adjacent instructions use results not yet produced/saved
- Branch hazards: you need to jump in the instruction flow but only figure out some number of cycles later

Hazards

- Data hazards: adjacent instructions use results not yet produced/saved



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