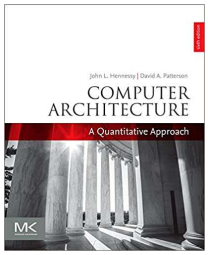
EE 156: Advanced Topics in Computer Architecture

Spring 2023
Tufts University

Instructor: Prof. Mark Hempstead
mark@ece.tufts.edu
Lecture 6: ISAs and Pipelining
[Appendicies A and C]

Resources

- Text: "Computer Architecture: A Quantitative Approach," **Sixth Edition**, Hennessy and Patterson



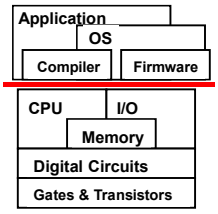
- Older editions are available in the library, but they use the MIPS ISA instead of RISC-V
- **Otherwise the key concepts are similar**

Lecture Outline

- ISA review
  - Overview
  - CISC vs. RISC
  - RISC-V ISA
- Implementation Review
- Introduction to Pipelining
- Hazards of Pipelining

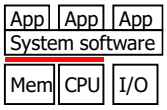
- Slide sources: David Brooks, David Patterson, Milo Martin,
- Additional References
  - Appendix A (ISAs)
  - Appendix C (Implementations and Pipelining)
  - "Computer Organization and Design: The Hardware/Software Interface" David A. Patterson and John L. Hennessy. ISBN: 978-0123747501

Instruction Set Architecture (ISA)



- What is an ISA?
  - A functional contract
- All ISAs similar in high-level ways
  - But many design choices in details
  - Two "philosophies": CISC/RISC
    - Difference is blurring
- Good ISA…
  - Enables high-performance
  - At least doesn't get in the way
- Compatibility is a powerful force
  - Tricks: binary translation, µISAs

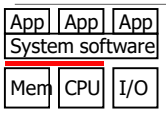
Program Compilation




```
int array[100], sum;
void array_sum() {
   for (int i=0; i<100;i++) {
      sum += array[i];
   }
}
```

- **Program** written in a "high-level" programming language
  - C, C++, Java, C#
  - Hierarchical, structured control: loops, functions, conditionals
  - Hierarchical, structured data: scalars, arrays, pointers, objects
- **Compiler**: translates program to **assembly**
  - Parsing and straight-forward translation
  - Compiler also optimizes
- Question: who compiled the compiler?
  - Answer: early compilers were written in assembly code!

Assembly & Machine Language

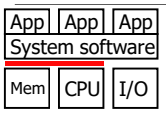


- **Assembly language**
  - Human-readable representation
- **Machine language**
  - Machine-readable representation
  - 1s and 0s (often displayed in "hex")
- **Assembler**
  - Translates assembly to machine

Machine code	Assembly code
x9A00	CONST R5, #0
x9200	CONST R1, array
xD320	HICONST R1, array
x9464	CONST R2, sum
xD520	HICONST R2, sum
x6640	LDR R3, R1, #0
x6880	LDR R4, R2, #0
x18C4	ADD R4, R3, R4
x7880	STR R4, R2, #0
x1261	ADD R1, R1, #1
x1BA1	ADD R5, R5, #1
x2B64	CMPI R5, #100
x03F8	BRn array_sum_loop

Example is in "LC4" a toy **instruction set architecture**, or **ISA**

Example Assembly Language & ISA

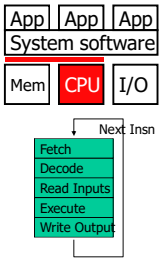


- **MIPS**: example of real ISA
  - 32/64-bit operations
  - 32-bit insns
  - 64 registers (32 integer, 32 FP)
  - ~100 different insns
  - Full OS support

Example code is MIPS, but all ISAs are similar at some level

```
    .data
array: .space 100
sum:   .word 0
    .text
array_sum:
    li $5, 0
    la $1, array
    la $2, sum
array_sum_loop:
    lw $3, 0($1)
    lw $4, 0($2)
    add $4, $3, $4
    sw $4, 0($2)
    addi $1, $1, 1
    addi $5, $5, 1
    li $6, 100
    blt $5, $6, array_sum_loop
```

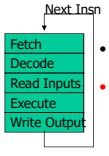

Instruction Execution Model



- The computer is just finite state machine
  - **Registers** (few of them, but fast)
  - **Memory** (lots of memory, but slower)
  - **Program counter** (next insn to execute)
    - Called "instruction pointer" in x86
- A computer executes **instructions**
  - **Fetches** next instruction from memory
  - **Decodes** it (figure out what it does)
  - **Reads** its **inputs** (registers & memory)
  - **Executes** it (adds, multiply, etc.)
  - **Write** its **outputs** (registers & memory)
  - **Next insn** (adjust the program counter)
- **Program is just "data in memory"**
  - Makes computers programmable ("universal")

The Sequential Model

- Basic structure of all modern ISAs
- **Program order**: total order on dynamic insns
- Convenient feature: **program counter (PC)**
  - What is a program counter?
  - Insn itself stored in memory at location pointed to by PC
  - Next PC is next insn unless insn says otherwise



- Processor logically executes loop at left
- **Atomic**: insn finishes before next insn starts
  - Can break this constraint physically (pipelining)
  - But must maintain illusion to preserve programmer sanity

What Makes a Good ISA?

- **Programmability**
  - Easy to express programs efficiently?
- **Implementability**
  - Easy to design high-performance implementations?
  - More recently
    - Easy to design low-power, high-reliability, low-cost implementations?
- **Compatibility**
  - Easy to maintain programmability (implementability) as languages and programs (technology) evolves?
  - x86 (IA32) generations: 8086, 286, 386, 486, Pentium, PentiumII, PentiumIII, Pentium4, Core2…
- **The goal** is to design an ISA that will do a good job on applications that won't be designed until 20 years from now ☺
  - 2nd best is to just design a reasonable ISA, and rely people's reluctance to change it.

Compilers

- Most assembly code is written by compilers. An ISA should make their job easy.
- Rules of thumb
  - Regularity: "principle of least astonishment"
  - Orthogonality
- Compilers must "schedule" instructions to maximize parallelization
  - ISA can make this easier by having latencies and register usages be straightforward.

Implementability

- Every ISA can be implemented
  - Not every ISA can be implemented efficiently
- Classic high-performance implementation techniques
  - Pipelining, parallel execution, out-of-order execution (more later)
- Certain ISA features make these difficult
  - Variable instruction lengths/formats: complicate decoding
  - Implicit state: complicates dynamic scheduling
  - Variable latencies: complicates scheduling
  - Difficult to interrupt instructions: complicate many things
    - Example: memory copy instruction

RISC-V ISA

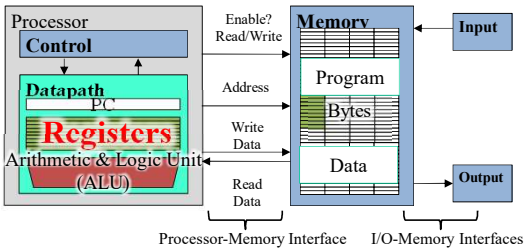
RISC-V ISA

- New fifth-generation RISC design from UC Berkeley
- Realistic & complete ISA, but open & small
- Not over-architected for a certain implementation style
- Both 32-bit (RV32) and 64-bit (RV64) address-space variants
- Designed for multiprocessing
- Efficient instruction encoding
- Easy to subset/extend for education/research
- Increasing momentum with industry adoption

Assembly Variables: Registers

- Unlike HLL like C or Java, assembly does not have *variables* as you know and love them
  - More primitive, closer what simple hardware can directly support
- Assembly operands are objects called registers
  - Limited number of special places to hold values, built directly into the hardware
  - Operations can only be performed on these!
- Benefit: Since registers are directly in hardware, they are very fast (faster than 1 ns - light travels 1 foot in 1 ns!!!)

Registers live inside the Processor



16

Number of RISC-V Registers

- Drawback: Since registers are in hardware, there are a limited number of them
  - Solution: RISC-V code must be carefully written to efficiently use registers
- 32 registers in RISC-V, referred to by number **`x0`** – **`x31`**
  - Registers are also given symbolic names, described later
  - Why 32? Smaller is faster, but too small is bad. Goldilocks principle ("This porridge is too hot; This porridge is too cold; this porridge is just right")
- Each RISC-V register is 32 bits wide (***RV32*** variant of RISC-V ISA)
  - Groups of 32 bits called a word in RISC-V ISA
  - P&H CoD textbook uses the 64-bit variant RV64 (explain differences later)
- **`x0`** is special, always holds value zero
  - So really only 31 registers able to hold variable values

17

17

C, Java Variables vs. Registers

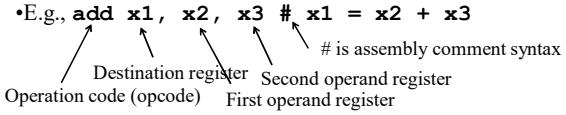
- In C (and most HLLs):
  - Variables declared and given a type
    - Example:
```
int fahr, celsius;
char a, b, c, d, e;
```
  - Each variable can ONLY represent a value of the type it was declared (e.g., cannot mix and match *int* and *char* variables)
- In Assembly Language:
  - Registers have no type;
  - Operation determines how register contents are interpreted

18

18

RISC-V Instruction Assembly Syntax

- Instructions have an opcode and operands
- E.g., `add x1, x2, x3 # x1 = x2 + x3`



Addition and Subtraction of Integers

- Addition in Assembly
  - Example: `add x1,x2,x3` (in RISC-V)
  - Equivalent to: a = b + c (in C)

    where C variables ⇔ RISC-V registers are:

    a ⇔ x1, b ⇔ x2, c ⇔ x3
- Subtraction in Assembly
  - Example: `sub x3,x4,x5` (in RISC-V)
  - Equivalent to: d = e - f (in C)

    where C variables ⇔ RISC-V registers are:

    d ⇔ x3, e ⇔ x4, f ⇔ x5

Addition and Subtraction of Integers Example 1

- How to do the following C statement?

  a = b + c + d - e;
- Break into multiple instructions

```
add x10, x1, x2 # a_temp = b + c
add x10, x10, x3 # a_temp = a_temp + d
sub x10, x10, x4 # a = a_temp - e
```

- A single line of C may turn into several RISC-V instructions

Immediates

- Immediates are numerical constants
- They appear often in code, so there are special instructions for them
- Add Immediate:

```
addi x3,x4,-10      (in RISC-V)
```
f = g - 10 (in C)

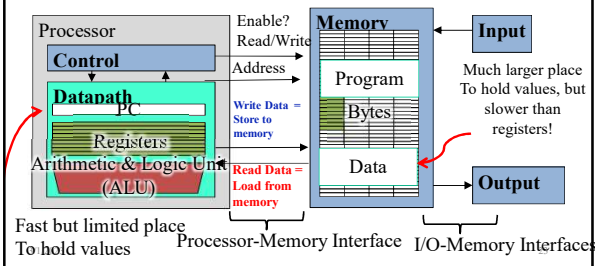
where RISC-V registers `x3,x4` are associated with C variables **f, g**

- Syntax similar to `add` instruction, except that last argument is a number instead of a register

```
add x3,x4,x0      (in RISC-V)
```
f = g (in C)

22

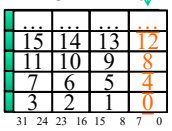
Data Transfer: Load from and Store to memory



Memory Addresses are in Bytes

- Data typically smaller than 32 bits, but rarely smaller than 8 bits (e.g., char type)–works fine if everything is a multiple of 8 bits
- 8 bit chunk is called a *byte* (1 word = 4 bytes)
- Memory addresses are really in *bytes*, not words
- Word addresses are 4 bytes apart
  - Word address is same as address of rightmost byte – least-significant byte (i.e. Little-endian convention)

Least-significant byte in a word



Least-significant byte gets the smallest address

3/1/2023

24

Transfer from Memory to Register

- C code
```
int  A[100];
g = h + A[3];
```
- Using Load Word (`lw`) in RISC-V:
```
lw   x10,12(x13)  # Reg x10 gets A[3]
add  x11,x12,x10  # g = h + A[3]
```

Note: `x13` – base register (pointer to A[0])
`12` – offset in bytes
Offset must be a constant known at assembly time

Transfer from Register to Memory

- C code
```
int  A[100];
A[10] = h + A[3];
```
- Using Store Word (`sw`) in RISC-V:
```
lw   x10,12(x13)   # Temp reg x10 gets A[3]
add  x10,x12,x10   # Temp reg x10 gets h + A[3]
sw   x10,40(x13)   # A[10] = h + A[3]
```

Note: `x13` – base register (pointer)
`12, 40` – offsets in bytes

x13+12 and x13+40 must be multiples of 4

Loading and Storing Bytes

- In addition to word data transfers (`lw`, `sw`), RISC-V has byte data transfers:
  - load byte: `lb`
  - store byte: `sb`
- Same format as `lw`, `sw`
- E.g., `lb x10,3(x11)`
  - contents of memory location with address = sum of "3" + contents of register x11 is copied to the low byte position of register x10.

RISC-V also has "unsigned byte" loads (`lbu`) which zero extend to fill register. Why no unsigned store byte `sbu`?

RISC-V Logical Instructions

- Useful to operate on fields of bits within a word
  - e.g., characters within a word (8 bits)
- Operations to pack /unpack bits into words
- Called *logical operations*

Logical operations	C operators	Java operators	RISC-V instructions
Bit-by-bit AND	&	&	**and**
Bit-by-bit OR	|	|	**or**
Bit-by-bit XOR	^	^	**xor**
Shift left logical	<<	<<	**sll**
Shift right logical	>>	>>	**srl**

28

Logical Shifting

- Shift Left Logical: `slli x11,x12,2` #x11=x12<<2
  - Store in x11 the value from x12 shifted 2 bits to the left (they fall off end), inserting 0's on right; << in C

Before: $0000\ 0002_{hex}$ $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}$

After: $0000\ 0008_{hex}$ $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1000_{two}$

What arithmetic effect does shift left have?

- Shift Right Logical: `srli` is opposite shift; >>
  - Zero bits inserted at left of word, right bits shifted off end

29

Arithmetic Shifting

- *Shift right arithmetic* (**srai**) moves *n* bits to the right (insert high-order sign bit into empty bits)
- For example, if register x10 contained

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 0111_{two} = -25_{ten}$
- If execute sra x10, x10, 4, result is:

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$
- Unfortunately, this is NOT same as dividing by 2^n
  - Fails for odd negative numbers
  - C arithmetic semantics is that division should round towards 0

30

Computer Decision Making

- Based on computation, do something different
- In programming languages: *if*-statement

- RISC-V: *if*-statement instruction is

  `beq register1,register2,L1`

  means: go to statement labeled L1
  if (value in register1) == (value in register2)
  ….otherwise, go to next statement
- **`beq`** stands for *branch if equal*
- Other instruction: **`bne`** for *branch if not equal*

31

Types of Branches

- **Branch** – change of control flow

- **Conditional Branch** – change control flow depending on outcome of comparison
  - branch *if* equal (**`beq`**) or branch *if not* equal (**`bne`**)
  - Also branch if less than (**`blt`**) and branch if greater than or equal (**`bge`**)

- **Unconditional Branch** – always branch
  - a RISC-V instruction for this: *jump (`j`)*

32

Example *if* Statement

- Assuming translations below, compile *if* block

  f → `x10` g → `x11` h → `x12`
  i → `x13` j → `x14`

```
if (i == j)              bne x13,x14,Exit
  f = g + h;             add x10,x11,x12
                              Exit:
```

- May need to negate branch condition

33

Example *if-else* Statement

- Assuming translations below, compile

  f → x10 g → x11 h → x12

  i → x13 j → x14

```
if (i == j)                    bne x13,x14,Else
  f = g + h;                   add x10,x11,x12
else                                 j Exit
  f = g - h;     Else:sub x10,x11,x12
                                Exit:
```


Magnitude Compares in RISC-V

- Until now, we've only tested equalities (== and != in C); General programs need to test < and > as well.
- RISC-V magnitude-compare branches:
  - "Branch on Less Than"
  - Syntax: `blt reg1,reg2, label`
  - Meaning: if (reg1 < reg2) // treat registers as signed integers goto label;
- "Branch on Less Than Unsigned"
  - Syntax: `bltu reg1,reg2, label`
  - Meaning: if (reg1 < reg2) // treat registers as unsigned integers goto label;

C Loop Mapped to RISC-V Assembly

```
int A[20];
int sum = 0;
for (int i=0; i<20;
i++)
   sum +=  A[i];
```

```
add x9, x8, x0 # x9=&A[0]
add x10, x0, x0 # sum=0
add x11, x0, x0 # i=0
Loop:
  lw x12, 0(x9) # x12=A[i]
  add x10,x10,x12 # sum+=
  addi x9,x9,4  # &A[i++]
  addi x11,x11,1 # i++
  addi x13,x0,20 # x13=20
  blt x11,x13,Loop
```

Additional RISC-V References

- Lectures on RISC-V from Berkeley:
  - `http://inst.eecs.berkeley.edu/~cs61c/fa17/`
  - https://inst.eecs.berkeley.edu/~cs61c/fa17/lec/05/
  - https://inst.eecs.berkeley.edu/~cs61c/fa17/lec/06/
  - https://inst.eecs.berkeley.edu/~cs61c/fa17/lec/07/
- RISC-V tech report and specification
  - https://riscv.org/technical/specifications/
  - https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.pdf
- Green sheet (available in textbook and Canvas)
  - https://inst.eecs.berkeley.edu/~cs61c/fa17/img/riscvcard.pdf

(Appendix C)

RISC-V IMPLEMENTATION AND INTRO TO PIPELINING

Implementation (Appendix C)

- The ISA describes the interface to the programmer/compiler
- But … the ISA does not describe how computation is completed in HW and how the memories and ALUs are connected
- ISAs are timing independent
  - Implementation (Microarchitecture) specifies timing

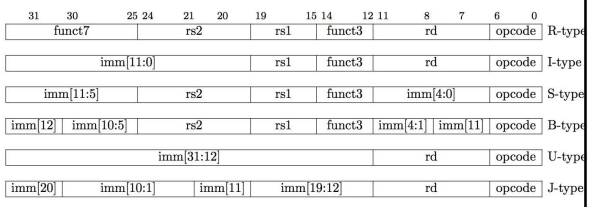
Instructions as Numbers (1/2)

- Most data we work with is in words (32-bit chunks):
  - Each register is a word
  - **`lw`** and **`sw`** both access memory one word at a time
- So how do we represent instructions?
  - Remember: Computer only understands 1s and 0s, so assembler string "`add x10,x11,x0`" is meaningless to hardware
  - RISC-V seeks simplicity: since data is in words, make instructions be fixed-size 32-bit words also
    - Same 32-bit instructions used for RV32, RV64, RV128

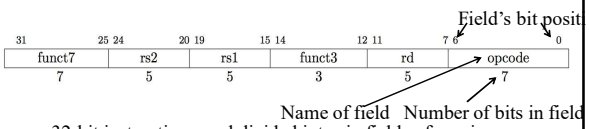
Instructions as Numbers (2/2)

- One word is 32 bits, so divide instruction word into "fields"
- Each field tells processor something about instruction
- We could define different fields for each instruction, but RISC-V seeks simplicity, so define six basic types of instruction formats:
  - R-format for register-register arithmetic operations
  - I-format for register-immediate arithmetic operations and loads
  - S-format for stores
  - B-format for branches (minor variant of S-format, called SB before)
  - U-format for 20-bit upper immediate instructions
  - J-format for jumps (minor variant of U-format, called UJ before)

Summary of RISC-V Instruction Formats

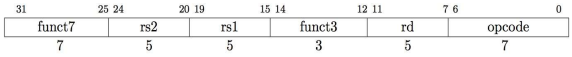


R-Format Instruction Layout



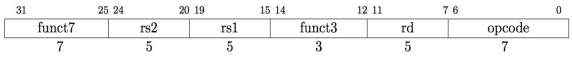
- 32-bit instruction word divided into six fields of varying numbers of bits each: 7+5+5+3+5+7 = 32
- Examples
  - opcode is a 7-bit field that lives in bits 6-0 of the instruction
  - rs2 is a 5-bit field that lives in bits 24-20 of the instruction

R-Format Instructions opcode/funct fields



- **opcode**: partially specifies what instruction it is
  - Note: This field is equal to 0110011_{two} for all R-Format register-register arithmetic instructions
- **funct7+funct3**: combined with **opcode**, these two fields describe what operation to perform
- Question: Why aren't **opcode** and **funct7** and **funct3** a single 17-bit field?
  - We'll answer this later

R-Format Instructions register specifiers

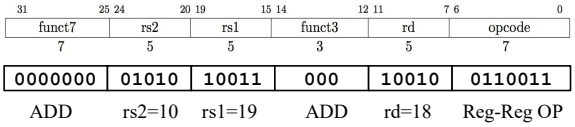


- **rs1** (Source Register #1): specifies register containing first operand
- **rs2** : specifies second register operand
- **rd** (Destination Register): specifies register which will receive result of computation
- Each register field holds a 5-bit unsigned integer (0-31) corresponding to a register number (**x0**-**x31**)

R-Format Example

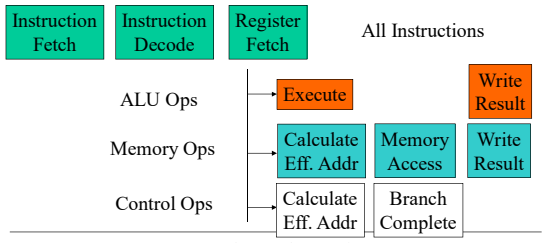
- RISC-V Assembly Instruction:

```
add  x18,x19,x10
```



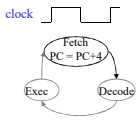
Implementation Review

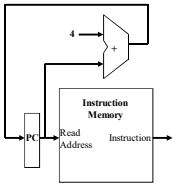
- First, let's think about how different instructions get executed



Fetching Instructions

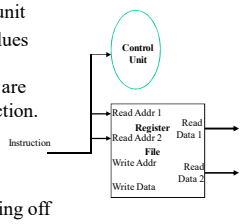
- Fetching instructions involves
  - reading the instruction from the Instruction Memory
  - updating the PC value to be the address of the next (sequential) instruction





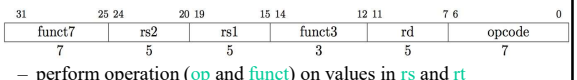
Decoding Instructions

- Decoding instructions involves
  - sending the fetched instruction's opcode and function field bits to the control unit
    - Reading one or two values from the register file.
    - Register-file addresses are contained in the instruction.
- What if an instruction only uses one operand?
  - Try to save power by turning off the clock to part of the register file.

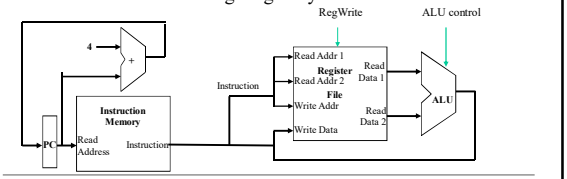


Executing R Format Operations

- R format operations (`add, sub, slt, and, or`)

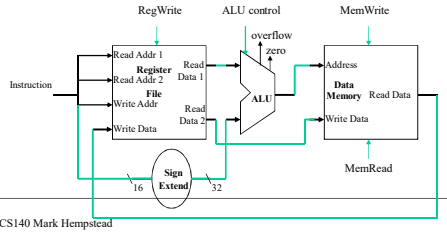


- perform operation (op and funct) on values in rs and rt
- store the result back into the register file (into location rd)
- Just assume the RF timing magically works.



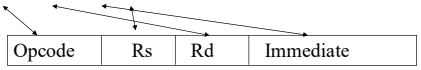
Executing Load and Store Operations

- Load and store operations involves
  - compute memory address by adding RF[rs]+immed; the base register plus a 16-bit signed-extended offset field in the instruction
  - store value (read from the Register File during decode) written to the Data Memory
  - load value, read from the Data Memory, written to the Register File



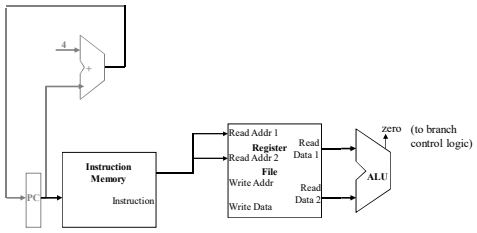
Calculate Effective Address: Memory Ops

- Calculate Memory address for data
- ALU_{output} <= A + Imm
- LW R10, 10(R3)



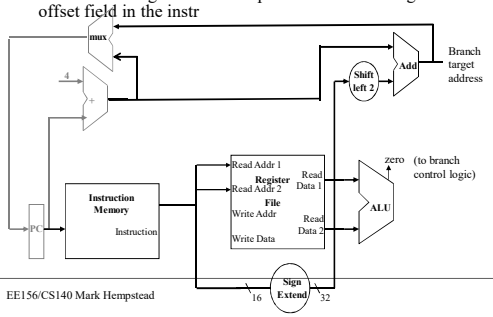
Executing Branch Operations

- Branch operations involves
  - Test the branch condition; e.g., compare two registers equality (**zero** ALU output)



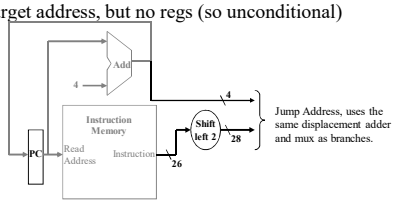
Executing Branch Operations

- Branch operations involves
  - Test the branch condition; e.g., compare two registers equality (**zero** ALU output)
  - Add branch-target address = updated PC +16-bit signed-extended offset field in the instr

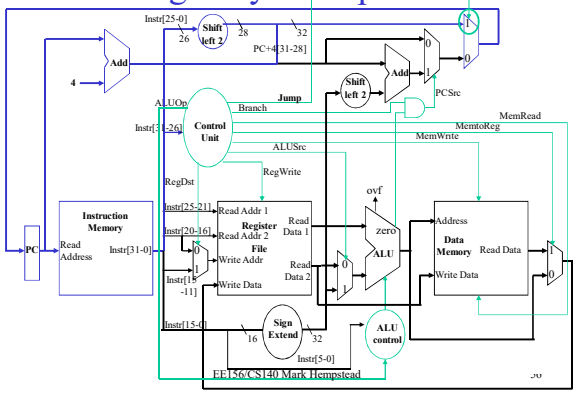


Executing Jump Operations

- Jump operation involves
  - replace the lower 28 bits of the PC with the lower 26 bits of the fetched instruction shifted left by 2 bits
  - Longer target address, but no regs (so unconditional)



Final Single-Cycle Implementation



How to make it faster?

- The design does not have flip-flops (pipeline registers)
- All work must complete in a single cycle
  - Cycle time is set by the longest instruction
  - Hardware will sit idle. (i.e. instruction fetch unit only used for a fraction of the clk cycle)
- Can we pipeline the work?
- Will we have any problems with that?
  - Yes, every problem that we had with our washer & dryer pipeline will crop up here again.

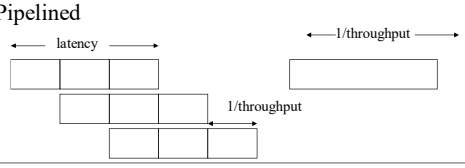
What is Pipelining?

- Implementation where multiple instructions are simultaneously overlapped in execution
  - Instruction processing has N different stages
  - Overlap different instructions working on different stages
- Pipelining is not new
  - Laundry – Washer/Dryer
  - Ford's Model-T assembly line
  - IBM Stretch [1962]
  - Since the '70s nearly all computers have been pipelined
- Concepts covered in Appendix A

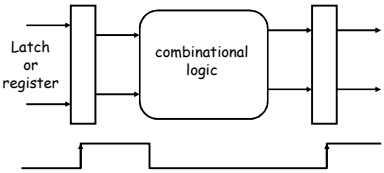
Pipelining Advantages

- Unpipelined

- Pipelined

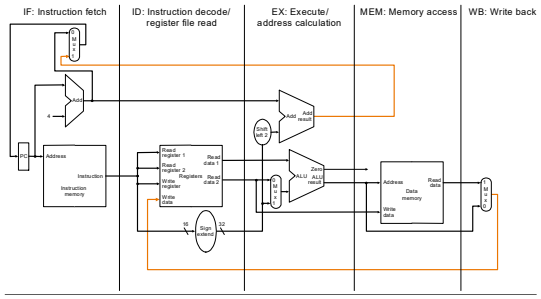


What's a Clock Cycle?

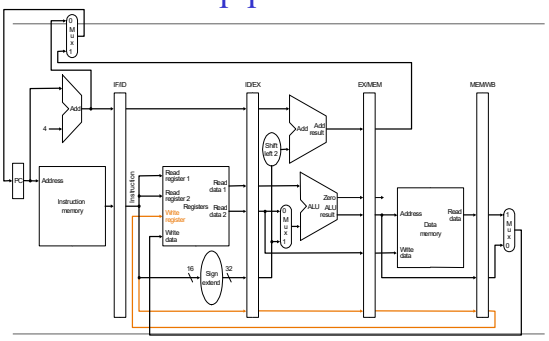


- All combinational logic must settle in one cycle.
- It gets harder with latches instead of flops (few people use latches nowadays)
- Skew and jitter on the clock wires make the accounting a bit harder also.

Recall from Earlier…



Now the pipelined version



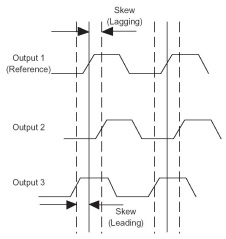
Ideal Pipelining Performance

- Assume instruction execution takes N stages; each stage #i takes time t_i.
- With no pipeline
  - Single Instruction latency, $T = \Sigma t_i$
  - Throughput = 1/T
  - Time to finish M instructions = M*T
- For an N-stage pipeline
  - Define t_c=max(Σt_i) … *introduces inefficiency unless all t_i are equal.*
  - Single Instruction latency = Nt_c
  - Throughput = $1/t_c$
  - Time to finish M instructions = (N+M-1) t_c
- $CPI_{Ideal} = \frac{CPI_{withoutpipeline}}{\text{Pipeline Depth}}$

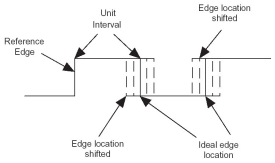
Costs of pipelining

- We greatly improved our throughput. But it wasn't for free
  - Single-instruction latency went from Σt_i to Nt_c.
  - Area got bigger (flops are not free!) → longer wires everywhere, more power.
  - Flops need clocks; clocks bounce a lot and consume substantial power.
  - Flops have delay. So each pipe stage adds some flop delay that doesn't do any actual computation, making the latency still worse.

Clock Skew and Jitter make pipelining even worse

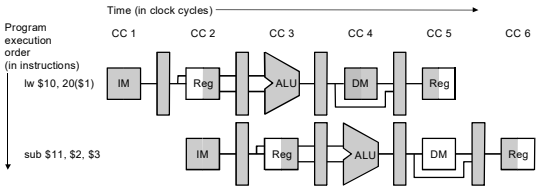


Clock Skew



Clock Jitter

Representation of Pipelines



lw X10, 20(X1)	IF	ID	EX	MEM	WB	
Sub X11, X2, X3		IF	ID	EX	MEM	WB

Pipeline Hazards

- Hazards: Situations that prevent the next instruction from executing in its designated clock cycle
  - Structural Hazards: when two different instructions want to use the same hardware resource in the same cycle (resource conflict)
  - Data Hazards: when an instruction depends on the result of a previous instruction that exposes overlapping of instructions
  - Control Hazards: pipelining of PC-modifying instructions (branch, jump, etc)

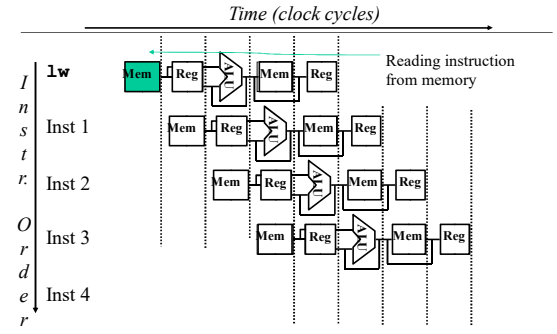
How to resolve hazards?

- Simple Solution: Stall the pipeline
  - Stops some instructions from executing
  - Make them wait for older instructions to complete
  - Simple implementation to "freeze" (de-assert write-enable signals on pipeline latches)
  - Inserts a "bubble" into the pipe
  - Must propagate upstream as well! Why?

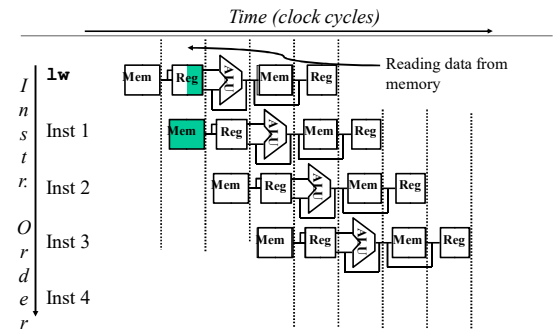
Structural Hazards

- Two cases when this can occur
  - Resource used more than once by the same instruction
  - Resource is not fully pipelined (FP Unit)
- Imagine that our pipeline shares I- and D-memory

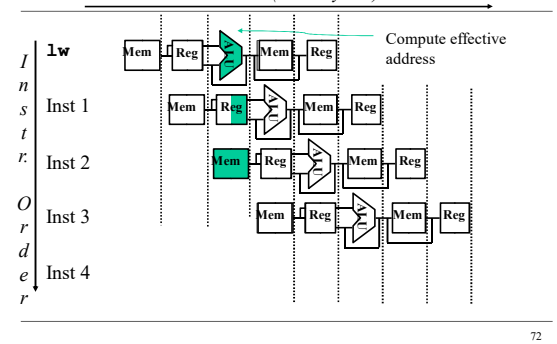
A Single Memory Would Be a Structural Hazard



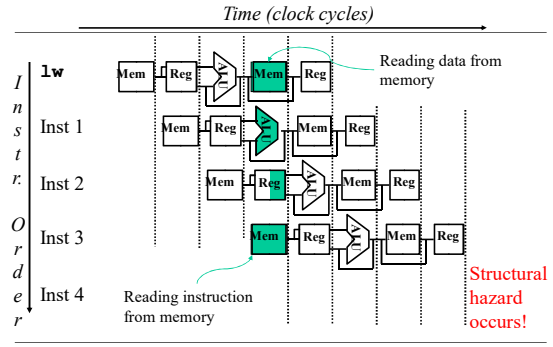
A Single Memory Would Be a Structural Hazard



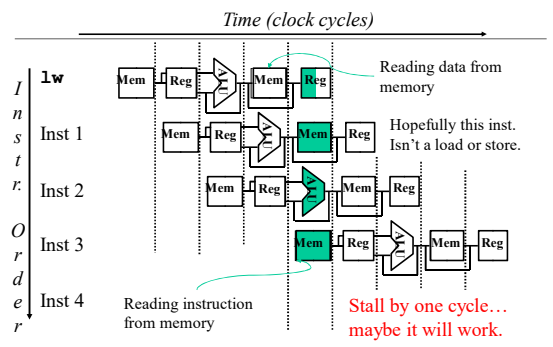
A Single Memory Would Be a Structural Hazard



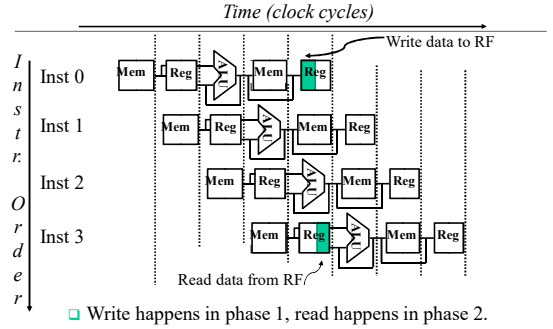
A Single Memory Would Be a Structural Hazard



A Single Memory Would Be a Structural Hazard



A Single RegFile is not a Structural Hazard



- Write happens in phase 1, read happens in phase 2.

Structural Hazards Solutions

- Stall
  - Low Cost, Simple (+)
  - Increases CPI (-)
  - Try to use for rare events in high-performance CPUs
    - Like lending out your last pair of clean pants
- Duplicate Resources
  - Decreases CPI (+)
  - Increases area, power, maybe cycle time (-)
  - Use for cheap resources, frequent cases
    - Such as? I-, D-caches, Separate ALU/PC adders, Reg File Ports

Structural Hazards Solutions

- Pipeline Resources
  - High performance (+)
  - Control is simpler than duplication (+)
  - Pipelining a divider is expensive (-)
  - Use when frequency makes it worthwhile
  - Ex. Fully pipelined FP add/multiplies critical for scientific
- Good news
  - Structural hazards don't occur as long as each instruction uses a resource
    - At most once
    - Always in the same pipeline stage
    - For one cycle
  - RISC ISAs are designed with this in mind, reduces structural hazards

Pipeline Stalls

- What could the performance impact of unified instruction/data memory be?

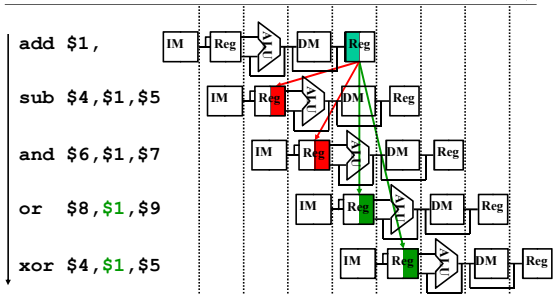
Loads ~15% of instructions, Stores ~10%

Prob (Ifetch + Dfetch) = .25

$CPI_{Real} = CPI_{Ideal} + CPI_{Stall} = 1.0 + .25 = 1.25$

Register Usage Can Cause Data Hazards

- Dependencies backward in time cause hazards



- Read after write data hazard

Solutions to RAW Hazards

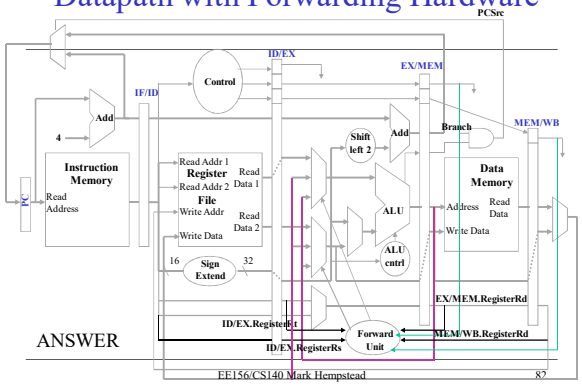
- Read After Write (RAW): As usual, we have a couple of choices
- Stall whenever we have a RAW
  - Huge performance penalty, dependencies are common!
- Use Bypass/Forwarding to minimize the problem
  - Data is ready by end of EXE (Add) or MEM (Load)
    - Add muxes to datapath to select proper value instead of regfile
    - Add control to track which regs will be written with which data, find hazards and drive the mux selects
    - Only stall when absolutely necessary

Forwarding, Bypassing

Cycle	1	2	3	4	5	6	7	8
Add R3, R2, R1	IF	ID	EX	MEM	WB			
Add R4, R3, R5		IF	ID	EX	MEM	WB		
Add R6, R3, R5			IF	ID	EX	MEM	WB	
Add R7, R3, R5				IF	ID	EX	MEM	WB

- Code is now "stall-free"
- Does the final "Add R7, R3, R5" need a bypass? No
- Are there any cases where we must stall? No

Datapath with Forwarding Hardware



ANSWER

Load Use Hazards

Cycle	1	2	3	4	5	6	7	8
lw R3, 10(R1)	IF	ID	EX	MEM	WB			
Add R4, R3, R5		IF	ID	EX	MEM	WB		
Add R6, R3, R5			IF	ID	EX	MEM	WB	

- Unfortunately, we can't forward "backward in time"

Cycle	1	2	3	4	5	6	7	8
lw R3, 10(R1)	IF	ID	EX	MEM	WB			
Add R4, R3, R5		IF	ID	EX/stall	EX	MEM	WB	
Add R6, R3, R5			IF	ID/stall	ID	EX	MEM	WB

Compiler tricks to avoid stalls

a = b +c;
d = e – f;

How many cycles for each?

No Scheduling Version	Scheduled Version
`LW Rb, b`	`LW Rb, b`
`LW Rc, c`	`LW Rc, c`
`ADD Ra, Rb, Rc`	`LW Re, e`
`SW Ra, a`	`ADD Ra, Rb, Rc`
`LW Re, e`	`LW Rf, f`
`LW Rf, f`	`SW Ra, a`
`SUB Rd, Re, Rf`	`SUB Rd, Re, Rf`
`SW Rd, d`	`SW Rd, d`

Cost of forwarding

- A stall increases latency and hurts throughput, so you improve both whenever you avoid a stall. But:
  - all of the extra muxes have delay, which costs latency and throughput every single cycle. Typically this is far outweighed by the fewer stalls.
  - The extra muxes, and the non-trivial control logic, all add area and power.
  - Long wires to forward data from one part of the die to another add substantial power.
  - If you're not careful, you might wind up inadvertently increasing t_c.

Pipe stalls and clock gating

- When the pipe is stalled, one or more pipe stages are doing nothing.
  - Mantra: "do nothing efficiently." How can we keep as few nodes bouncing as possible?
  - Typical answer: gate off the clock for the entire pipe stage.
- Critical paths may prevent this
  - When one pipe stage stalls, then all upstream stages must also stall.
  - The pipe is geographically big, and upstream stages may be far away from the stall source.
  - It may not be easy to tell the upstream stages in time!

Control Hazards

Cycle	1	2	3	4	5	6	7	8
Branch Instr.	IF	ID	EX	MEM	WB			
Instr +1		IF	ID	EX	MEM	WB		
Instr +2			IF	ID	EX	MEM	WB	MEM

- Does this work?
- No: if we branch on Rs==Rt, we use the ALU and don't know whether to branch or not until the end of EX.
- Simple solution – stall until outcome is known

Control Hazards

Cycle	1	2	3	4	5	6	7	8
Branch Instr.	IF	ID	EX	MEM	WB			
Instr +1		IF/stall	IF/stall	IF	ID	EXE	MEM	WB
Instr +2					IF	ID	EXE	MEM

- Does this work?
- No. The branch does not write the PC until its WB stage. But Instr+1 is doing its IF during the branch's MEM stage, when the PC is not yet written.

Control Hazards

Cycle	1	2	3	4	5	6	7	8
Branch Instr.	IF	ID	EX	MEM	WB			
Instr +1		IF/stall	IF/stall	IF/stall	IF	ID	EXE	MEM
Instr +2						IF	ID	EXE

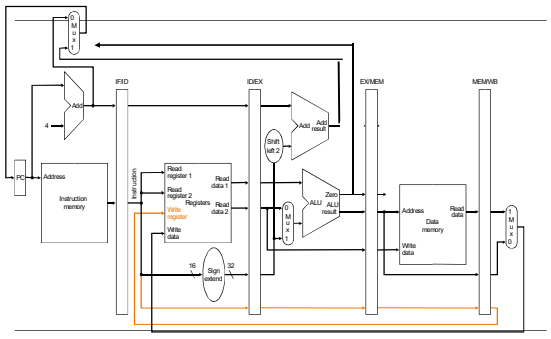
- Does this work?
- Finally, yes. But Instr+1 had to stall for 3 cycles.
- Length of control hazard is branch delay
  - In this simple case, it is 3 cycles (assume 10% cond. branches)
  - $CPI_{Real} = CPI_{Ideal} + CPI_{Stall} = 1.0 + 3 \text{ cycles} * .1 = 1.3$
  - Can be worse, with deeper pipes & branch-heavy code

Control Hazards: Solutions Fast Branch Resolution

- Fast Branch Resolution
  - Cut off a pipe stage, now write the PC in the branch's MEM stage.
  - See picture on the next foil.
  - Now there's just a 2-cycle branch stall.

Cycle	1	2	3	4	5	6	7	8
Branch Instr.	IF	ID	EX	MEM	WB			
Instr +1		IF/stall	IF/stall	IF	ID	EX	MEM	WB
Instr +2					IF	ID	EXE	MEM

Send new PC & zero back faster



Control Hazards: Solutions Fast Branch Resolution

- Even faster branch resolution
  - Make the branch decision at the end of ID (only works for simple conditions like testing if reg>0).
  - Adder in ID for PC + immediate targets
  - Write the PC in EX
  - The critical paths are usually too big to work, and we still have a 1-cycle stall.

Cycle	1	2	3	4	5	6	7	8
Branch Instr.	IF	ID	EX	MEM	WB			
Instr +1		IF/stall	IF	ID	EX	MEM	WB	
Instr +2				IF	ID	EX	MEM	WB

Control Hazards: Branch Characteristics

- Integer Benchmarks: 14-16% instructions are conditional branches
- FP: 3-12%
- On Average:
  - 67% of conditional branches are “taken”
  - 60% of forward branches are taken
  - 85% of backward branches are taken
  - Why?

Control Hazards: Solutions

1. Stall Pipeline
   - Simple, No backing up, No Problems with Exceptions
2. Assume not taken
   - Speculation requires back-out logic:
     - What about exceptions, auto-increment, etc
   - Bets the "wrong way"
3. Assume taken
   - Doesn't help in simple pipeline! (don't know target)
4. Delay Branches
   - Can help a bit… we'll see pro's and con's soon
- Another Option (Preview) …. predict branches

Control Hazards: Assume Not Taken

Cycle	1	2	3	4	5	6	7	8
Untaken Branch	IF	ID	EX	MEM	WB			
Instr +1		IF	ID	EX	MEM	WB		
Instr +2			IF	ID	EX	MEM	WB	

Looks good if we're right!

Cycle	1	2	3	4	5	6	7	8
Taken Branch	IF	ID	EX	MEM	WB			
Instr +1		IF	flush	flush	flush	flush		
Branch Target				IF	ID	EX	MEM	WB
Branch Target +1					IF	ID	EX	MEM

But, we have to roll back and flush the pipeline if we are wrong!

CPUs and Pipelines Summary

- Executing instructions can be split into distinct logic stages
  - Basic MIPS is Fetch, Decode, Execute, Mem, Writeback
- Pipelining can share resources across multiple instructions
- Pipelining adds complexity and costs
  - Data dependancies can cause data hazards

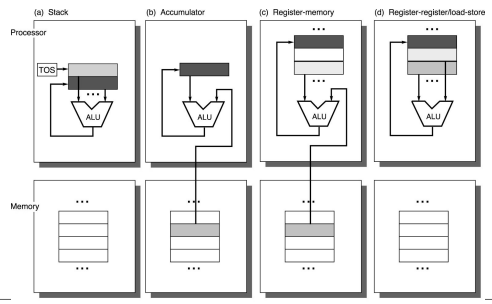
Backup

- This slides are for your reference but will not be tested in class
- The slides cover material from the textbook that you might be asked on interviews
  - Compiler Optimizations
  - ISA classification
  - Stack based ISAs (in embedded systems and specialized (e.g. Intel Multimedia) systems)
- CISC vs RISC
  - Historically interesting, through the Turing Lecture covers this as well

Compiler Optimizations

- Primarily reduce instruction count
  - Eliminate redundant computation, keep more things in registers
    - Registers are faster, fewer loads/stores
    - An ISA can make this difficult by having too few registers
- But also… (more talk a bit about these later on)
  - Reduce branches, jumps, and cache misses
  - Reduce dependences between nearby insns
    - An ISA can make this difficult
- Compiler optimizations are limited
  - Proebsting's Law: compiler optimization = order of magnitude less performance gain than hardware
  - This is 4X – but 4X performance is still substantial!
- Bottom Line: don't design an inefficient ISA and hope the compiler will fix everything.

Classifying ISAs



Stack

- Architectures with implicit "stack"
  - Acts as source(s) and/or destination, TOS is implicit
  - Push and Pop operations have 1 explicit operand
- Example: C = A + B
  - Push A // S[++TOS] = Mem[A]
  - Push B // S[++TOS] = Mem[B]
  - Add // Tem1 = S[TOS--], Tem2 = S[TOS--] , S[++TOS] = Tem1 + Tem2
  - Pop C // Mem[C] = S[TOS--]
- x86 FP uses stack. Why might this complicate pipelining?

Accumulator

- Architectures with one implicit register
  - Acts as source and/or destination
  - One other source explicit
- Example: C = A + B
  - Load A // (Acc)umulator <= A
  - Add B // Acc <= Acc + B
  - Store C // C <= Acc
- x86 uses accumulator concepts for integer
- Again, why might this complicate pipelining?

Register

- Most common approach
  - Fast, temporary storage (small)
  - Explicit operands (register IDs)
- Example: C = A + B

Register-memory	load/store
Load R1, Mem[A]	Load R1, Mem[A]
Add R3, R1, Mem[B]	Load R2, Mem[B]
Store R3, Mem[C]	Add R3, R1, R2
	Store R3, Mem[C]

- All RISC ISAs are load/store
- IBM 360, Intel x86, Moto 68K are register-memory

Operator Types

Type	Example
Arithmetic/Logical	add, subtract, and, or
Data transfer	load-stores
Control	branches, jump, procedure call, return, traps
System	OS call, virtual memory management
Floating point	FP add, subtract …
Decimal	Decimal add, multiply …
String	String compare, string move
Graphics	Pixel and vertex operations

Common Addressing Modes

Register	Add R5,R4, R3
Immediate	Add R5,R4, #3
Register Indirect	Load R4, (R1)
Base/Displacement	Load R4, 100(R1)
Indexed	Load R4, (R1+R2)
Direct	Load R4, (1001)
Memory Indirect	Load R4, @(R3)
Auto-increment	Load R4, (R2)+
Scaled	Load R4, (R2+4*R3)

Where might you use each of these?

A Language Analogy for ISAs

- Communication
  - Person-to-person → software-to-hardware
- Similar structure
  - Narrative → program
  - Sentence → insn
  - Verb → operation (add, multiply, load, branch)
  - Noun → data item (immediate, register value, memory value)
  - Adjective → addressing mode
- Many different languages, many different ISAs
  - Similar basic structure, details differ (sometimes greatly)
- Key differences between languages and ISAs
  - Languages evolve organically, many ambiguities, inconsistencies
  - ISAs are explicitly engineered and extended, unambiguous

THE RISC VS. CISC DEBATE

CISC vs. RISC

- Debate raged from early 80s through 90s
- Now it is fairly irrelevant
- Despite this Intel (x86 => Itanium) and DEC/Compaq (VAX => Alpha) have tried to switch
- Research in the late 70s/early 80s led to RISC
  - IBM 801 -- John Cocke – mid 70s
  - Berkeley RISC-1 (Patterson)
  - Stanford MIPS (Hennessy)

x86

- Variable length ISA (1-16 bytes)
- FP Operand Stack
- 2 operand instructions (extended accumulator)
  - Register-register and register-memory support
- Scaled addressing modes

- Has been extended many times (as AMD did with x86-64)
- Intel, initially went to IA64, now backtracked (EM64T)

RISC vs. CISC Arguments

- RISC
  - Simple Implementation
    - Load/store, fixed-format 32-bit instructions, efficient pipelines
  - Lower CPI
  - Compilers do a lot of the hard work
    - MIPS = Microprocessor without Interlocked Pipelined Stages
- CISC
  - Simple Compilers (assists hand-coding, many addressing modes, many instructions)
  - Code Density

After the dust settled

- Turns out it doesn't matter much
- Can decode CISC instructions into internal "micro-ISA"
  - This takes a couple of extra cycles (PLA implementation) and a few hundred thousand transistors
  - In 20 stage pipelines, 55M tx processors this is minimal
  - Pentium 4 caches these micro-Ops
- Actually may have some advantages
  - External ISA for compatibility, internal ISA can be tweaked each generation (Transmeta)

Compatibility

- In many domains, ISA must remain compatible
  - IBM's 360/370 (the *first* "ISA family")
  - Another example: Intel's x86 and Microsoft Windows
    - x86 one of the worst designed ISAs EVER, but survives
- **Backward compatibility**
  - New processors supporting old programs
    - Can't drop features (**caution in adding new ISA features**)
    - Or, update software/OS to emulate dropped features (slow)
- **Forward (upward) compatibility**
  - Old processors supporting new programs. How?
    - Include a "CPU ID" so the software can test of features
    - Add ISA hints by overloading no-ops (example: x86's PAUSE)
    - New firmware/software on old processors to emulate new insn

The Compatibility Trap

- Easy compatibility requires forethought
  - Temptation: use some ISA extension for 5% performance gain
  - Frequent outcome: gain diminishes, disappears, or turns to loss
    - **Must continue to support gadget for eternity**
  - Example: "register windows" (SPARC)
    - Adds difficulty to out-of-order implementations of SPARC
- Compatibility trap door
  - How could you rid yourself of some ISA mistake in the past?
  - Make old instruction an "illegal" instruction on new machine
  - Operating system handles exception, emulates instruction, returns
    - Slow unless extremely uncommon for all programs