Introduction

- Course goals:

  Understanding of computer architecture, structure and evolution

- Computer architecture = instruction set architecture plus computer organization

- Instruction set architecture:

  Conceptual structure and functional behaviour of computing system as seen by programmer

- Computer organization:

  Physical implementation, described in terms of functional units, their interconnection, how information flow among them is controlled

Why We Teach This Course

- Understanding of what's inside the computer
- Introduction to material in future courses
- Architecture issues influence programming

Example: small code changes, big performance differences

```
#include<stdio.h>
#define NR 10000
#define NC 10000

int a[NR][NC];

void main() {
  int i,j;
  for (i=0;i<NR;i++){
    for (j=0;j<NC;j++){
      a[i][j]=32767; } } }
```

- Row-by-row (a[i][j]): 1.693 sec
- By column (a[j][i]): 27.045 sec
  (approx 16 times slower!)

Example on i7-2677M

```
#include<stdio.h>
#define NR 10000
#define NC 10000

int a[NR][NC];

void main() {
  int i,j;
  for (i=0;i<NR;i++){
    for (j=0;j<NC;j++){
      a[i][j]=32767; } } }
```

- Row-by-row (a[i][j]): 0.30 sec
- By column (a[j][i]): 1.24 sec
  (approx 4 times slower!)

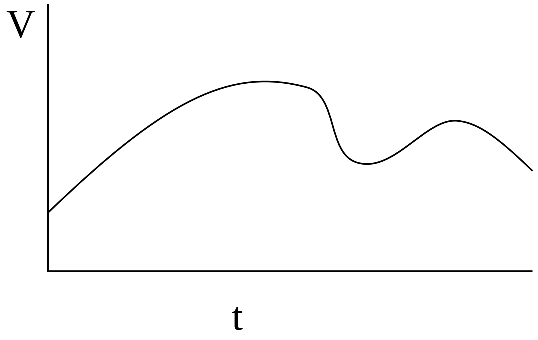
Copyright figure removed

A Brief Look at Electricity

- Computers work with current/voltage

$$V = IR$$

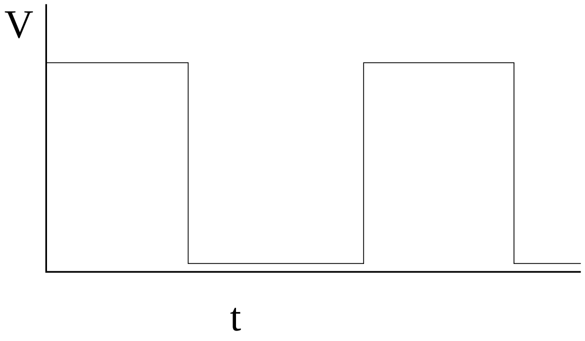
- These quantities are continuous
  Plot: voltage vs time



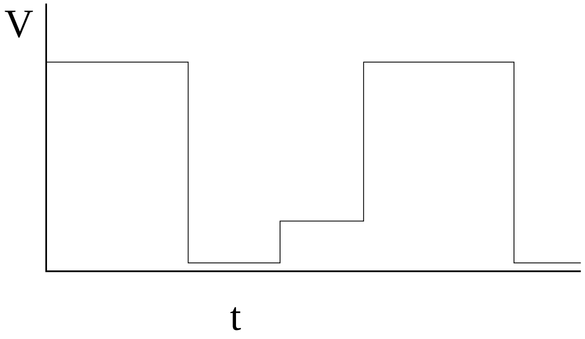
- They can be manipulated, but accuracy is difficult

Digitizing

- Discrete Signal



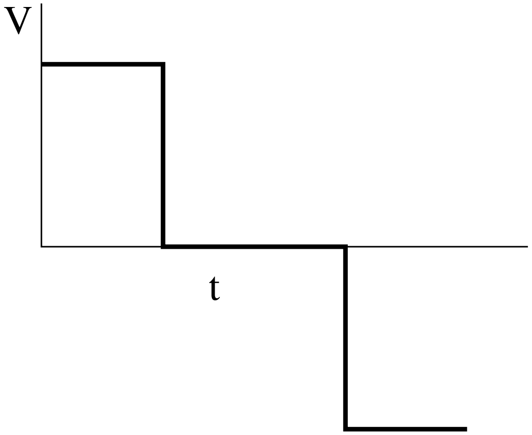
- Signal is either high (1) or low (0)
- Transformation could lead to intermediate values



but these can be “designed out”

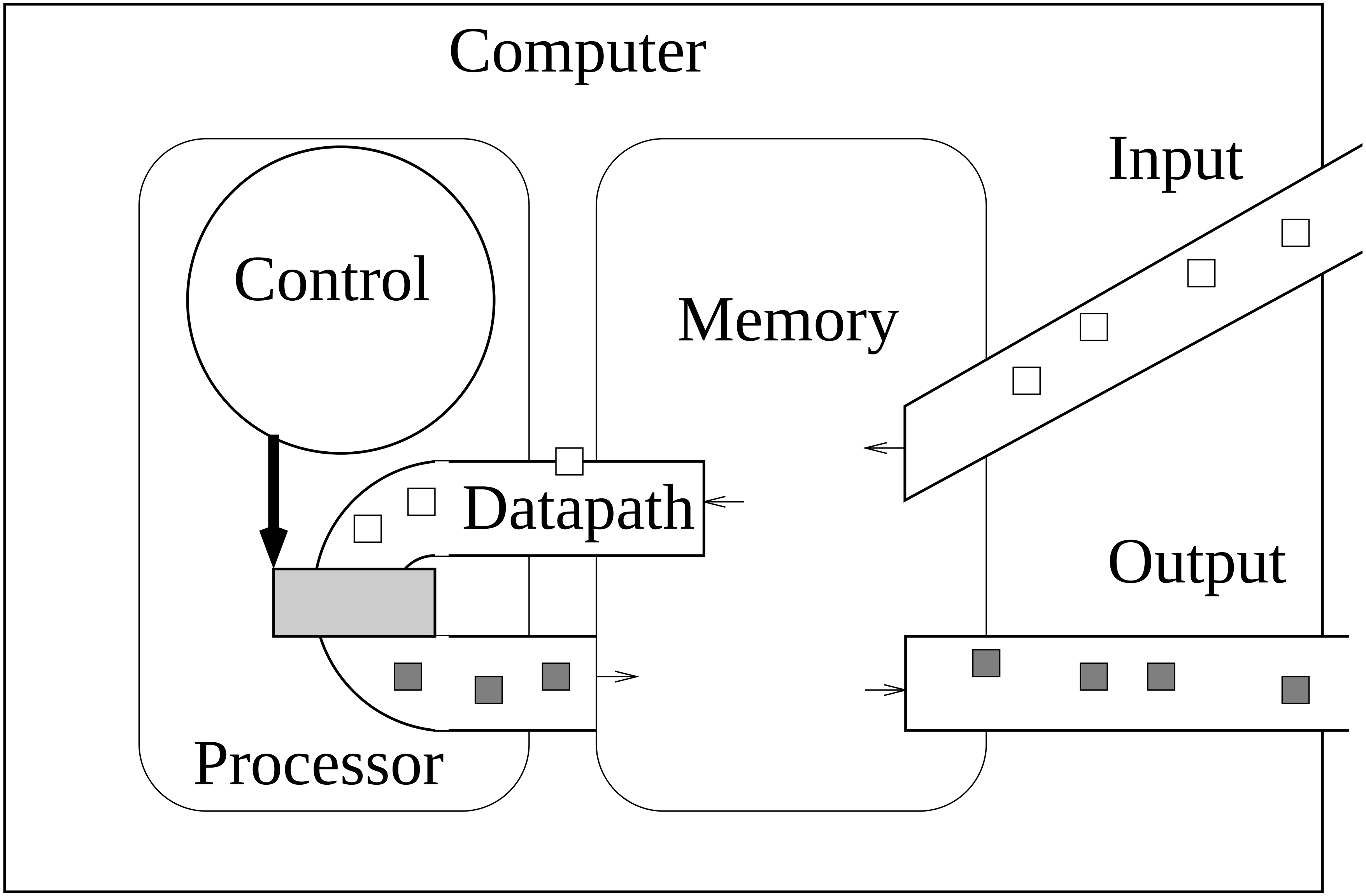
Why Binary?

- Could have more levels...



- Plus, Zero, Minus (+1, 0, -1 — ternary)
- Two levels are simpler and just as expressive
- Nearly all computers today use binary
- Nearly all computers have similar underlying structure

Computer Organization: The Big Picture



Course outline

- MIPS review, brief discussion of performance
- digital logic design
- data representation and manipulation
- designing a datapath
- single-cycle control unit
- multiple-cycle control units (hardwired and microprogrammed)
- pipelining and hazards
- memory hierarchies (caches and virtual memory)
- input/output
- multiprocessor systems
- case studies: VAX, SPARC, Pentium

CS 251 topics continued in other courses

- CS 240: memory management
- CS 350: operating systems
- CS 343: concurrency
- CS 370: scientific computation
- CS 454: distributed systems
- CS 456: networks

Logistics

- Course notes (lecture slides):
  - copies of slide text only (whitespace reduced)
  - not comprehensive
  - no substitute for lectures
- Textbooks for reference and additional study:
  - "Computer Organization and Design", David Patterson and John Hennessy, Revised 5th edition, 2014. **REQUIRED**
  - "Digital Design and Computer Architecture", David Harris and Sarah Harris, 2nd edition, 2012.
- 6 assignments (plus A0), midterm, final exam
- Course Webpage:
  `http://student.cs.uwaterloo.ca/˜cs251`
- Course newsgroup:
  `https://piazza.com/class#fall2016/cs251`

Assignments

- 6 assignments plus A0

  A0 not required, but worth one bonus mark
- Submit via Crowdmark

  Scan to submit
- Solutions in display case outside MC 4065

  Book authors request that solutions not be online

Slide

Copyright Issues

- Course notes contain figures from book
- We have copyright permission to include them, but unable to put course notes on the web.
- Assignments contain figures from the text
  Unable to put solutions on the web

Excessive Collaboration

- In the past, as many as 10%-20% of students in the course have been caught and penalized for excessive collaboration.
- Previous terms: encouraged to talk/discuss, but must write up solutions on own without checking with other students.

  Excessive similiaries treated as excessive collaboration.
- This term (F16): Collaboration fine. Hand in own copy of assignment.
- **Caution:** Doing assignments is critical for learning the material:

  > "I feel lost sometimes in lectures, but the assignments help a lot."
- **Caution:** Reading course text is critical for learning the material.

- What's not allowed:
  - Looking for solutions on the internet
  - Using solutions from previous terms
  - Photocopying (!) another student's assignment
  - Word-for-word copying
- Standard Penalty for first offense at Waterloo: no marks on the assignment and a deduction of 5% from the course grade, letter to associate dean.

  Additional penalties may apply depending on marking scheme.
- Standard Penalty for second offense: suspension for one term.

MIPS Review

- Computers execute assembly instructions

  In binary on computer, but text form for people

- Only simple operations

  Addition, subtraction, goto, conditional goto

- Instructions operate on two types of data
  - Registers—high speed access
  - RAM—slow to access

- This course uses MIPS, which we review here

  Optional homework assignment on MIPS (bonus point)

Registers

- There are 32 registers

  Can use like a variable in a program, but via MIPS instruction

  - Each register has 32 bits, four bytes
  - $0, $1,...,$31
  - Sometimes: $s0,...,$s7, $t0,...,$t7

    Either okay, but don't mix $1 and $s1 in same program!
  - $0 always contains 0

Instruction

Three general types of MIPS instructions

Format refers to how many and what type of operands

- R-Format: `add $1,$2,$3`

  Adds contents of \$2 to contents of \$3; store result in \$1

  Often written as `add rd,rs,rt`, where `rd` is the *destination* register

- I-Format: `addi $1, $2, 100`

  Adds *immediate* value 100 to contents of \$2; store result in \$1

- J-Format: `j 28`

  Used for branching; discussed later

Memory

- MIPS program can access 4 Giga-Bytes (4GB) of random access memory
- Memory accessed with number from 0 to $2^{32} - 1$
- Usually grouped in 4-byte blocks called *words*

  Most memory accesses are to addressed that are multiple of 4
- Both MIPS program and data are stored in memory

Program in memory

- Each program instruction is one word in length
- Instruction address is multiple of four
- Often write memory program as memory address followed by instruction:

Memory	
Address	Instruction
100:	`add $1,$2,$3`
104:	`sub $1,$3,$5`
108:	`addi $2,$12,16`

- Often don't need address and use symbolic label of important instructions:

```
start: add $1,$2,$3
       sub $1,$3,$5
       addi $2,$12,16
```

Control flow

- In MIPS, no conditional statements like `if`
- In MIPS, no loop constructions like `for, while`
- Control flow handled by goto-like commands
  - jump (unconditional goto)
  - `beq` (conditional goto)
- Special register, *program counter* (PC), stores address of executing instruction
- When non-goto instruction executed, PC incremented by 4
  This auto-increment advances the program to the next instruction

Jump instruction

- jump: `j`

```
100: j 28
104: add $1, $2, $3
108: sub $1, $3, $5
112: addi $2, $12, 16
```

- When jump executed, PC set to four times immediate argument

Conditional branch

- Example:

```
beq $1,$2,100
```

  - Compare contents of $1 to contents of $2
    If equal, add 4 times constant (100) and add to PC
    (×4 because constant is a word/instruction offset)
  - PC will also have 4 added to it
    PC updated to PC$+(4\times 100)+4$
  - If registers not equal, the instruction following branch is executed
- Constant can be negative
- `bne` similar but branches if values in registers are not equal

Conditional branch example

```
100:  add $1, $0, $0
104:  addi $2, $0, 6
108:  addi $1, $1, 5
112:  addi $2, $2, -1
116:  bne $2, $0, -3
120:  add $4, $6, $8
```

- Assume PC starts with value 100
- Fifth instruction (116) is conditional branch

  If contents of $2 not equal to zero, we branch to...

Memory access

- 32 registers clearly not enough to store data of most programs
- Special MIPS instructions to access 4GB RAM
- All memory accesses handled by two I-Format instruction
- *Load word*: Reads word from memory, stores in register

  ```
  100:  lw $1, 100($2)
  ```

  Read value stored at memory address 100+$2 (`M[100+$2]`) store result in register $1
- *Store word*: Takes value of register and write it to memory

  ```
  100:  sw $1, 100($2)
  ```

  Write the value in register $1 to `M[100+$2]`
- This is word address, so 100+$2 must be multiple of 4

Example revisited: Registers

```
#include<stdio.h>
#define NR 10000
#define NC 10000

int a[NR][NC];

void main() {
  int i,j;
  for (i=0;i<NR;i++){
    for (j=0;j<NC;j++){
      ; } } }
```

- `register int i,j`: 0.044 sec
- `int i,j`: 0.27 sec
  (approx 6 times slower!)

MIPS vs ARM

- MIPS and ARM assembly similar:

MIPS	**ARM**
`lw $1, 0($2)`	`LDR r1, [r2]`
`add $1,$2,$3`	`ADD r1,r2,r3`
`addi $1,$2,22`	`ADD r1,r2,#22`
`add $1,$0,$2`	`MOV r2,r1`

- ARM has 15 registers, MIPS has 32
  ARM: r15 is PC, r14 for subroutine calls
  MIPS: $0 is always 0
- ARM takes about 1/4 the transitors as MIPS
- ARM has conditional forms of instructions
  ARM compiler takes advantage of this; gcc does not

Performance

- Readings: skim Chapter 1, read section 1.6
- How can we compare different computer designs?
- Two important measures of performance:
  - Response time: time between start and completion of a task
  - Throughput: total amount of work done in a given time
- Improving response time usually improves throughput
- Analogy: grocery-store checkout
- Nearly all computers have a clock
- Useful concepts: clock ticks, cycles, rate

Some factors affecting response time

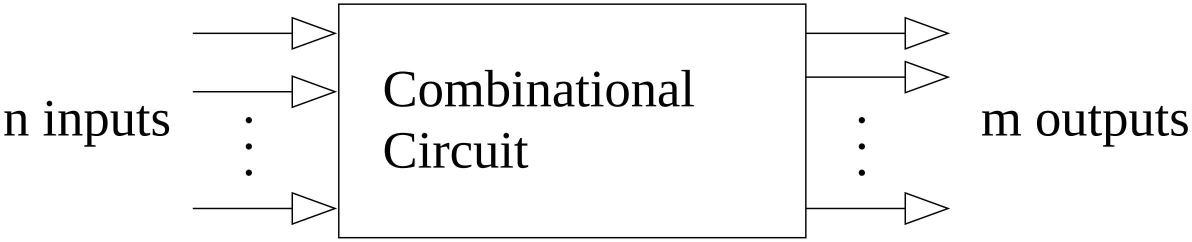
- Speed of clock
- Complexity of instruction set
- Efficiency of compilers
- Mix of instructions needed to complete a task
- Some choices in designing computers:
  - simple instruction set, fast clock, one instruction executed per clock cycle
  - complex instruction set, slower clock, one instruction executed per clock cycle
  - complex instruction set, faster clock, some instructions take multiple cycles to execute

Benchmarks

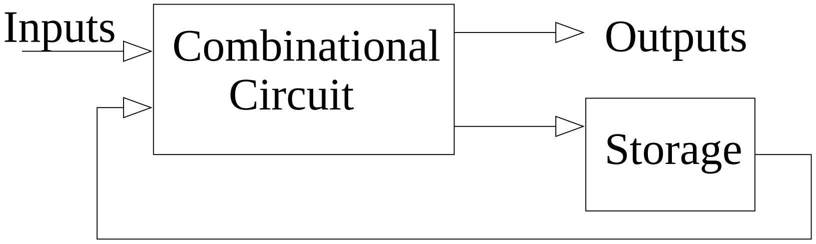
- Standard set of programs (and data) chosen to measure performance
- Advantages:
  - Provides basis for meaningful comparisons
  - Design by committee may eliminate vendor bias
- Disadvantages:
  - Vendors can optimize for benchmark performance
  - Possible mismatch between benchmark and user needs
  - Still an artificial measurement

Logic Blocks

- Readings: Appendix B, sections B.1–B-3, B.7-10.
- Combinational: without memory



- Sequential: with memory



- Inputs and outputs are 1/0
  (High/low voltage, true/false)

Specifying input/output behaviour

- Truth table: specifies outputs for each possible input combination

X	Y	Z	F	G
0	0	0	0	1
0	0	1	1	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	0

- Complete description, but big and hard to understand

Compact alternative: Boolean algebra

- Variables (usually A, B, C or X, Y, Z) have values 0 or 1
- OR ($+$) operator has result 1 iff either operand has value 1
- AND ($\cdot$) operator has result 1 iff both operands have value 1

  $A \cdot B$ often written AB
- NOT ($\neg$) operator has result 1 iff operand has value 0

  $\neg A$ usually written $\bar{A}$

OR		
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

AND		
A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

NOT	
A	¬A
0	1
1	0

- For truth table on previous slide, clearly $G = \overline{XYZ}$
- $F = \bar{X}\bar{Y}Z + X\bar{Y}Z + XY\bar{Z} + XYZ$ (not obvious)

Truth Table to Formula Using Minimal Terms

A	B	C	F	$\bar{A}\bar{B}C$	$A\bar{B}C$	ABC	$\bar{A}\bar{B}C+A\bar{B}C+ABC$
0	0	0	0	0	0	0	0
0	0	1	1	1	0	0	1
0	1	0	0	0	0	0	0
0	1	1	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	0	1	0	1
1	1	0	0	0	0	0	0
1	1	1	1	0	0	1	1

Two-Level Representations

- Any Boolean function can be represented as a sum of products (OR of ANDs) of literals
- Each term in sum corresponds to a single line in truth table with value 1
- This can be simplified by hand or by machine
- Product of sums representation may also be useful

Don’t Cares in Truth Tables

- Represented as X instead of 0 or 1
- When used in output, indicates that we don’t care what output is for that input
- When used in input, indicates outputs are valid for all inputs created by replacing X by 0 or 1 (useful in compressing truth tables)
- Example:

A	B	C	F
0	0	X	0
0	1	X	1
1	X	X	X

Compressed Truth Tables and Non-Minimal Terms

A	B	C	F	$\bar{A}\bar{B}C$	AC	$\bar{A}\bar{B}C+AC$
0	0	0	0	0	0	0
0	0	1	1	1	0	1
0	1	X	0	0	0	0
1	X	0	0	0	0	0
1	X	1	1	0	1	1

Using Overlapping Non-Minimal Terms

A	B	C	F	AB	AC	$AB+AC$
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	0	0	0	0
1	0	1	1	0	1	1
1	1	0	1	1	0	1
1	1	1	1	1	1	1

Laws of Boolean Algebra

Rule	Dual Rule	
$\overline{\overline{X}} = X$		
$X + 0 = X$	$X \cdot 1 = X$	(identity)
$X + 1 = 1$	$X \cdot 0 = 0$	(zero/one)
$X + X = X$	$XX = X$	(absorption)
$X + \overline{X} = 1$	$X\overline{X} = 0$	(inverse)
$X + Y = Y + X$	$XY = YX$	(commutative)
$X + (Y + Z) =$ $(X + Y) + Z$	$X(YZ) = (XY)Z$	(associative)
$X(Y + Z) = XY + XZ$	$X + YZ =$ $(X + Y)(X + Z)$	(distributive)
$\overline{X + Y} = \overline{X} \cdot \overline{Y}$	$\overline{XY} = \overline{X} + \overline{Y}$	(DeMorgan)

Formula Simplification Using Laws

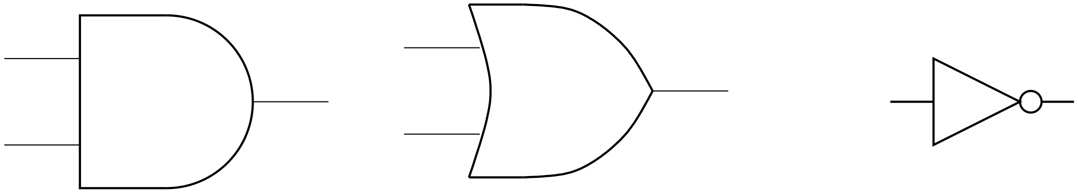
- We can use algebraic manipulation (based on laws) to simplify formulas
- An example using the previous truth table

$$\begin{aligned} F &= \overline{X}\,\overline{Y}Z + X\overline{Y}Z + XY\overline{Z} + XYZ \\ &= \overline{Y}Z(\overline{X} + X) + XY(\overline{Z} + Z) \\ &= \overline{Y}Z + XY \end{aligned}$$

- Difficult even for humans, tricky to automate
- Seems inherently hard to get “simplest” formula
- Is simplest formula the best for implementation?

Using Gates in Logic Design

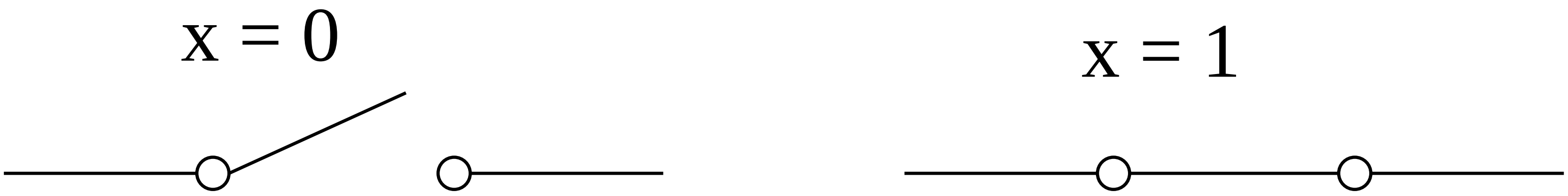
- Here are symbols for AND, OR, NOT gates



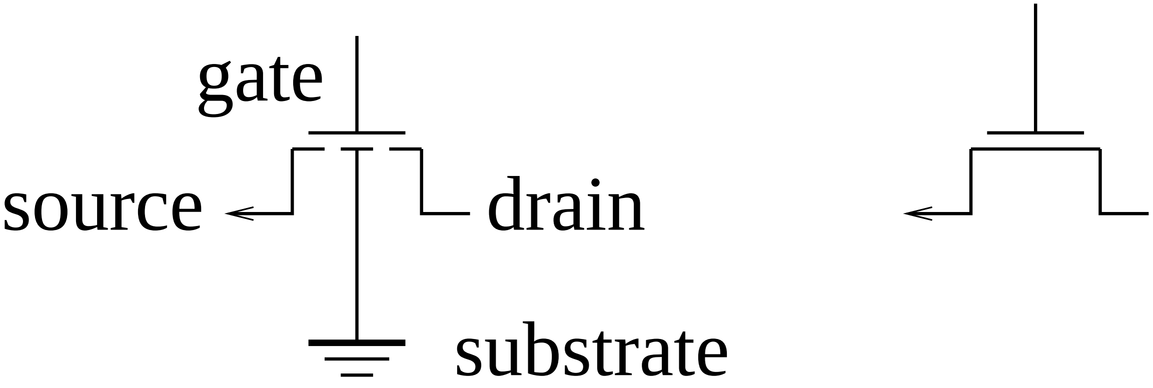
- NOT often drawn as "bubble" on input or output
- AND, OR can be generalized to many inputs (useful)
- We can design using AND, OR, NOT, and optimize afterwards
- In practice, logic minimization software works with NAND or NOR gates, or at transistor level

Implementing Gates Using Transistors

- Transistor: an electrically-controlled switch

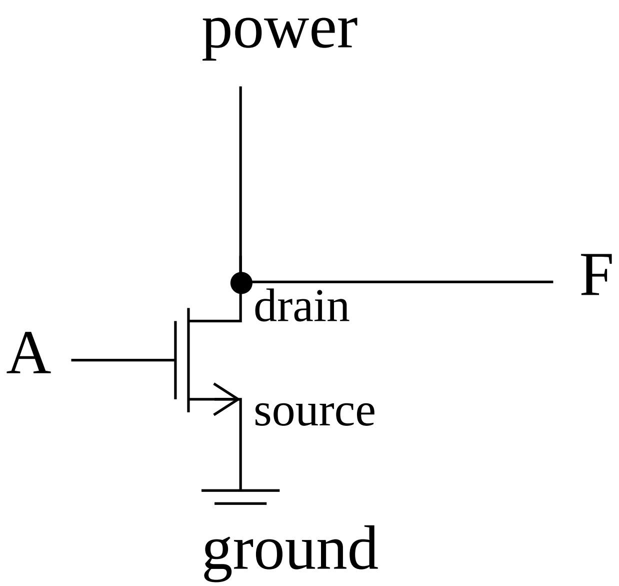


- An NMOS transistor ("n-transistor") and its symbol



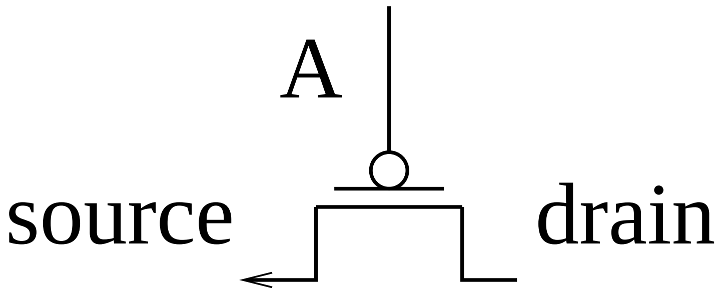
- This behaves like the switch above
- Problem: transmits strong 0 but weak 1

An NMOS NOT



- If $A = 1$, then low resistance between drain and source ($F = 0$)
- If $A = 0$, then very high resistance between drain and source ($F = 1$)
- Problem: in $A = 1$ case, lots of current flow

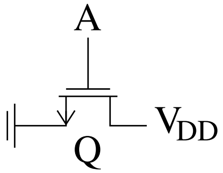
A PMOS transistor

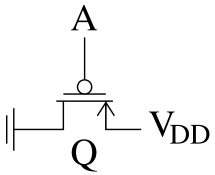


- Opposite behaviour to NMOS:
  - If $A = 1$, high resistance between drain and source
  - If $A = 0$, low resistance between drain and source
  - Transmits strong 1 but weak 0
- Denote inversion with “bubble”

Transistor Summary

- Two types of transistors: nmos, pmos





NMOS

Input	$A = 0$	$A = 1$
Resistance	High	Low

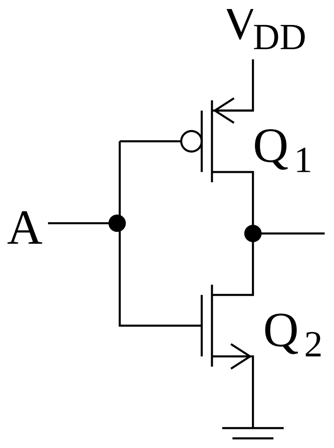
PMOS

Input	$A = 0$	$A = 1$
Resistance	Low	High

- To analyze CMOS circuit:
  - Make table with inputs, transistors, and output(s)
  - For each row of table (setting of inputs), check whether transistor resistance is High,Low
  - For each row of table, check if output has clean path to
    power (1)
    ground (0)

CMOS

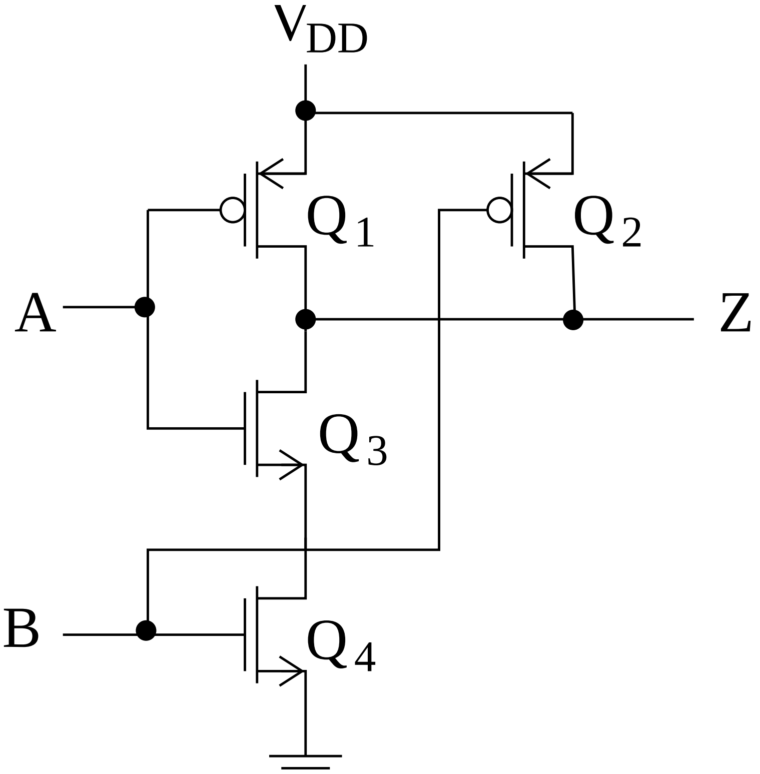
- CMOS circuits use both n-transistors and p-transistors
- Will build circuits with “clean” paths to exactly one of power and ground.
- CMOS NOT:



A	Q_1	Q_2	F
0	Low	High	1
1	High	Low	0

- No bad flow of current from power to ground
- No weak transmissions

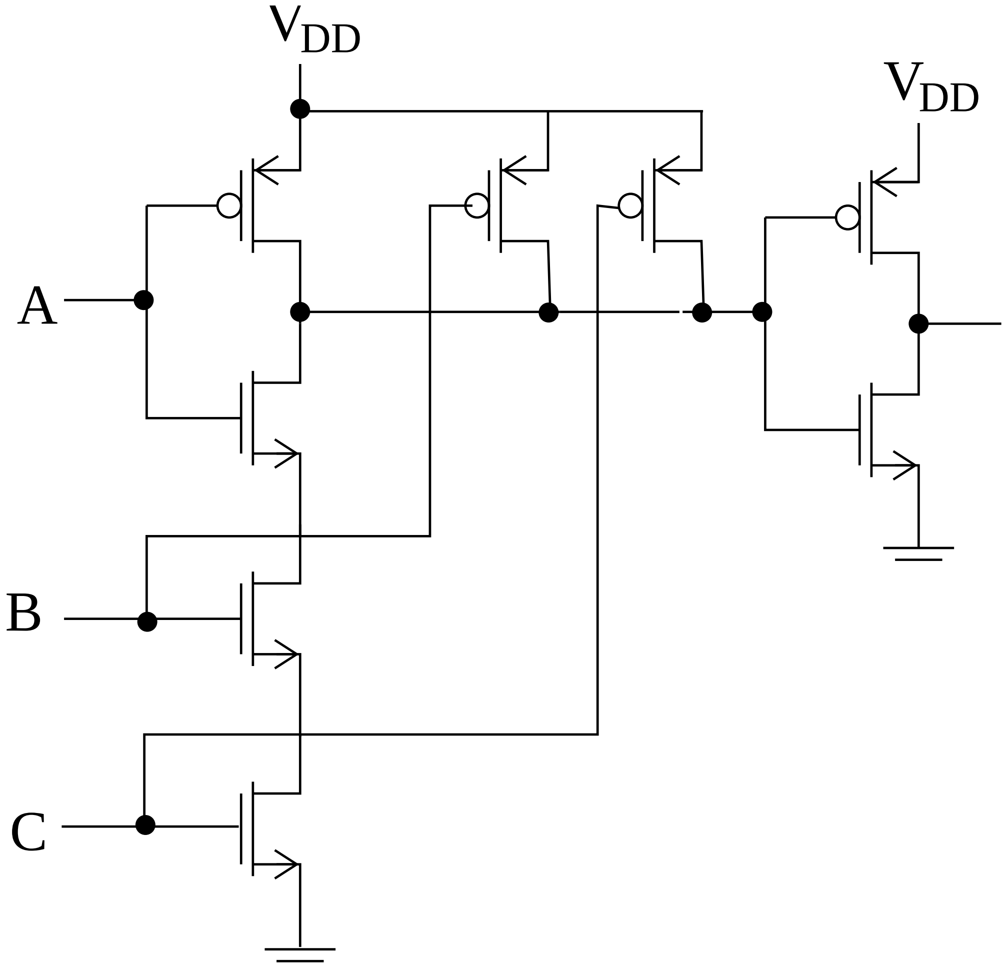
CMOS NAND



A	B	Q_1	Q_2	Q_3	Q_4	Z
0	0	Low	Low	High	High	1
0	1					
1	0					
1	1					

CMOS AND and OR

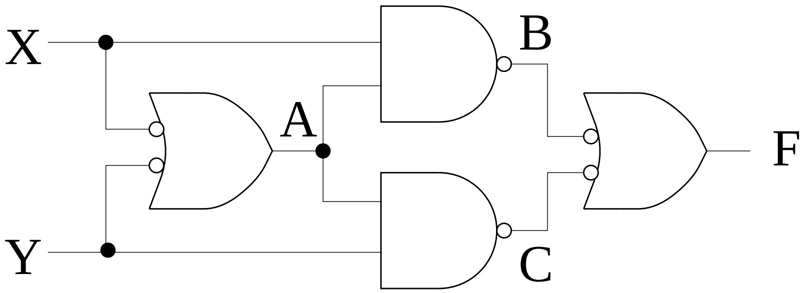
- To get AND and OR, add inverter at end
- Example: 3 Input AND



- Thus, NAND is preferred to AND in actual circuits

Deriving Truth Table from Circuit

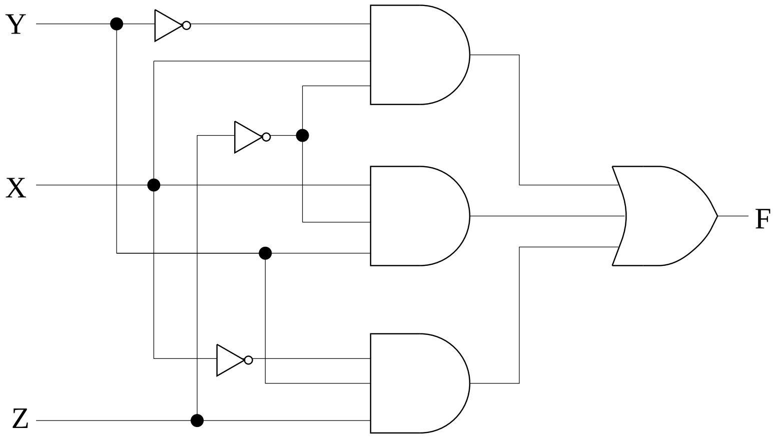
- Label intermediate gate outputs
- Fill in truth table in appropriate order



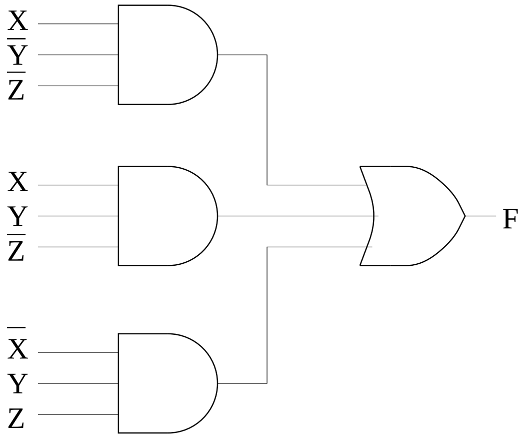
X	Y	A	B	C	F
0	0				
0	1				
1	0				
1	1				

Good Style in Circuit Drawing

- Assume all literals (variables and their negations) are available
- Rectilinear wires, dots when wires split
- Do not draw spaghetti wires for inputs; instead, write each literal as needed



Bad



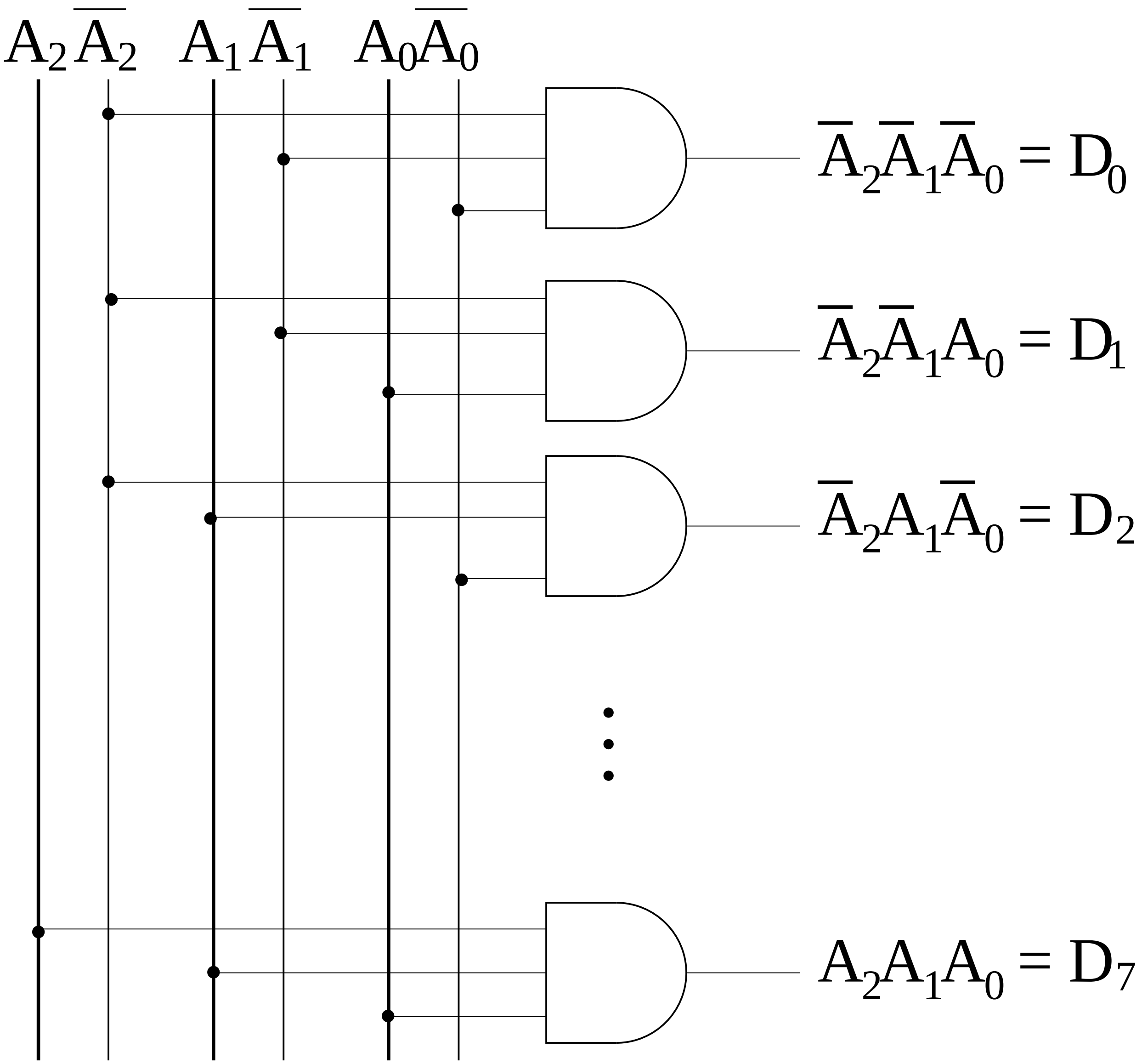
Good

Useful Components: Decoders

- n inputs, 2^n outputs (converts binary to “unary”)
- Example: 3-to-8 (or 3-bit) decoder

A_2	A_1	A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

- Circuit has regular structure

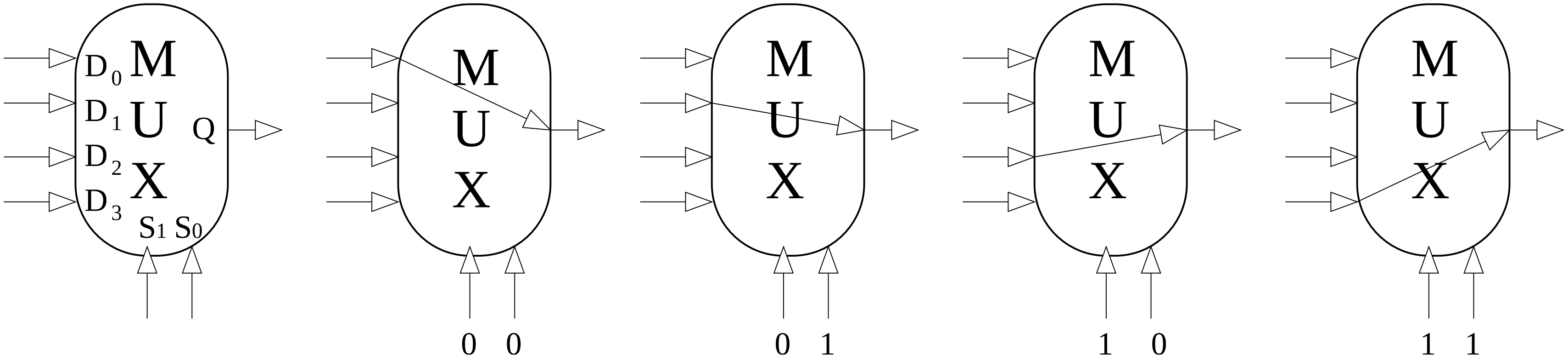

A_2 $\overline{A}_2$ A_1 $\overline{A}_1$ A_0 $\overline{A}_0$
$\overline{A}_2\overline{A}_1\overline{A}_0 = D_0$
$\overline{A}_2\overline{A}_1A_0 = D_1$
$\overline{A}_2A_1\overline{A}_0 = D_2$
$A_2A_1A_0 = D_7$

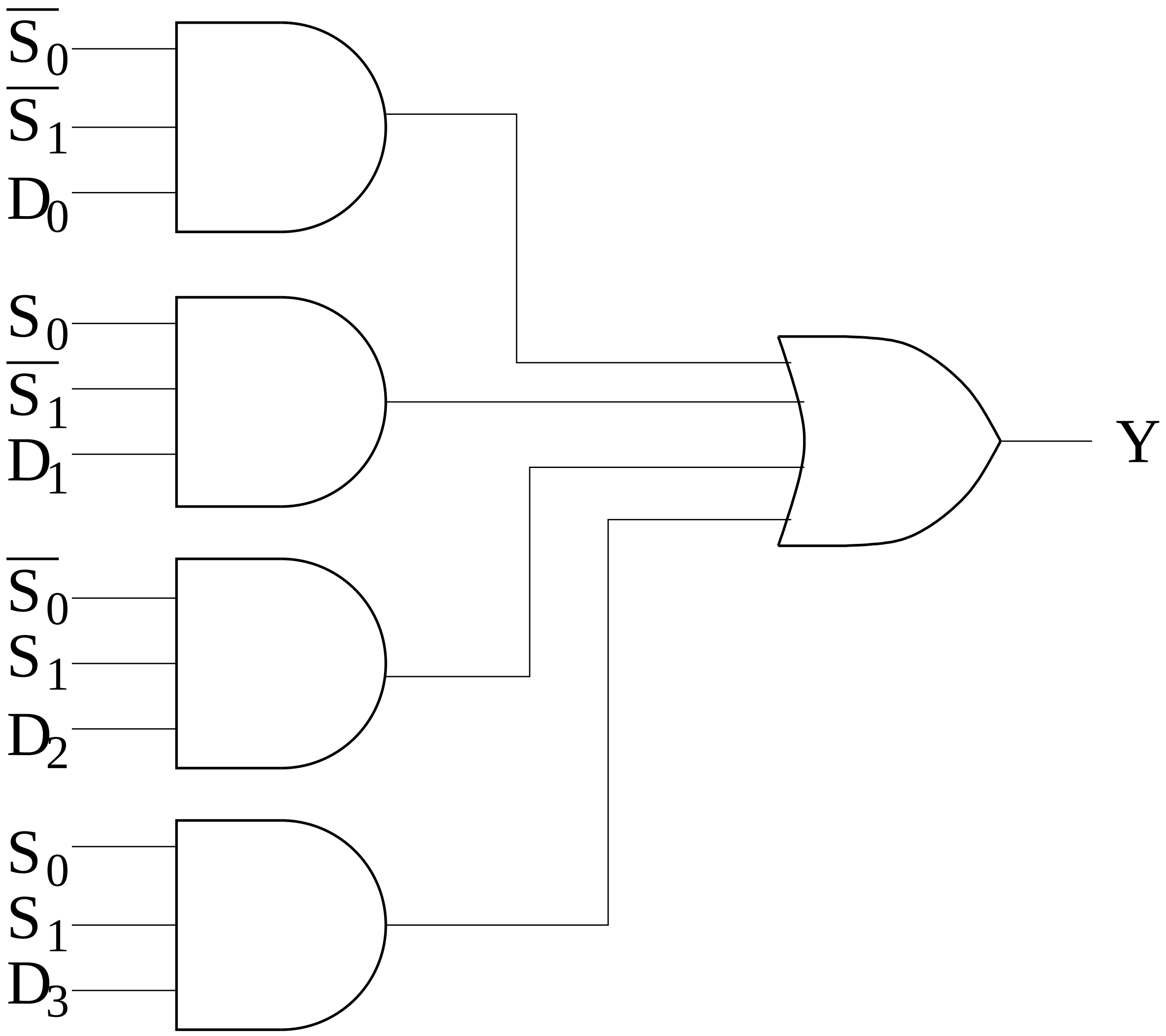
3-to-8 (or 3-bit) Decoder

Multiplexors

- Inputs: 2^n lines $(D_0, \ldots, D_{2^n-1})$
  n select lines $(S_{n-1}, \ldots, S_0)$
- Output: The value of the D_S line
- Example: 4-1 Multiplexer

S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3





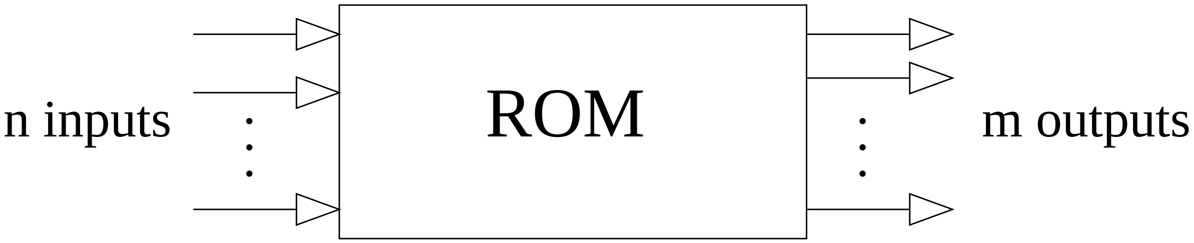
4-1 Multiplexor

Arrays of Logic Elements

- “Slash” notation is used to indicate lines carrying multiple bits, and to imply parallel constructions

Copyright figure removed

Implementing Boolean Functions: ROMs



- Can think of ROM as table of 2^n m-bit words
- Can think of ROM as implementing m one-bit functions of n variables
- Internally, consists of a decoder plus an OR gate for each output
- Types of ROM: PROM, EPROM, EEPROM
- PLAs - simplified ROM

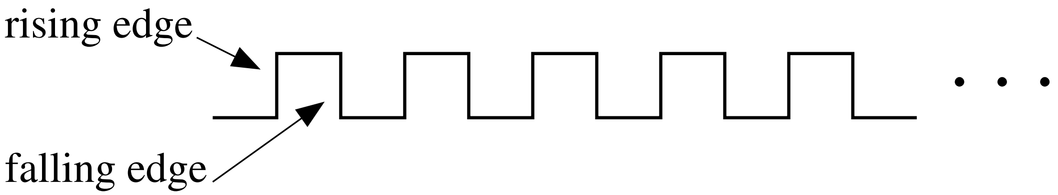
  Less hardware, but less flexible

Clocks and Sequential Circuits

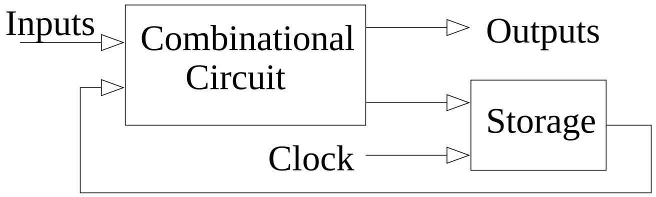
- Two types of sequential circuits:
- Synchronous: has a clock

  Memory changes only at discrete points in time

  Clock pulse:



  Block diagram:



  Easier to analyze, tend to be more stable

- Asynchronous: no clock

  Potentially faster and less power-hungry, but harder to design and analyze

SR Latch with NOR gates

- SR Latch with NOR gates

Copyright figure removed

- Problem: Behavior depends on *previous* values of Q and $\overline{Q}$ when $S = R = 0$
- Need to add time, talk about transitions

S,R transition	Q, $\overline{Q}$ transition
1,0 → 0,0	→
0,1 → 0,0	→
1,1 → 0,0	→

Functional Description of SR Latch

S	R	Q	$\overline{Q}$	
0	0	Q	$\overline{Q}$	Latch state (no change)
0	1	0	1	Reset state
1	0	1	0	Set state
1	1	?	?	Undefined

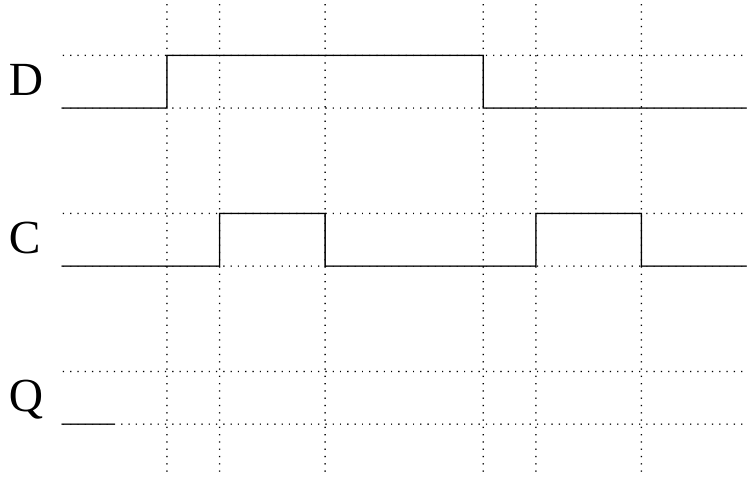
- Advantages:
  - Can “remember” value
  - Natural “reset” and “set” signals (SR=01 is “reset” to 0, SR=10 is “set” to 1)
- Disadvantages:
  - SR=11 input has to be avoided
  - No notion of a clock or change at discrete points in time yet

The D Latch

Copyright figure removed

C	D	Next state of Q
0	X	No change
1	0	$Q = 0$ (Reset)
1	1	$Q = 1$ (Set)

Graphical example:

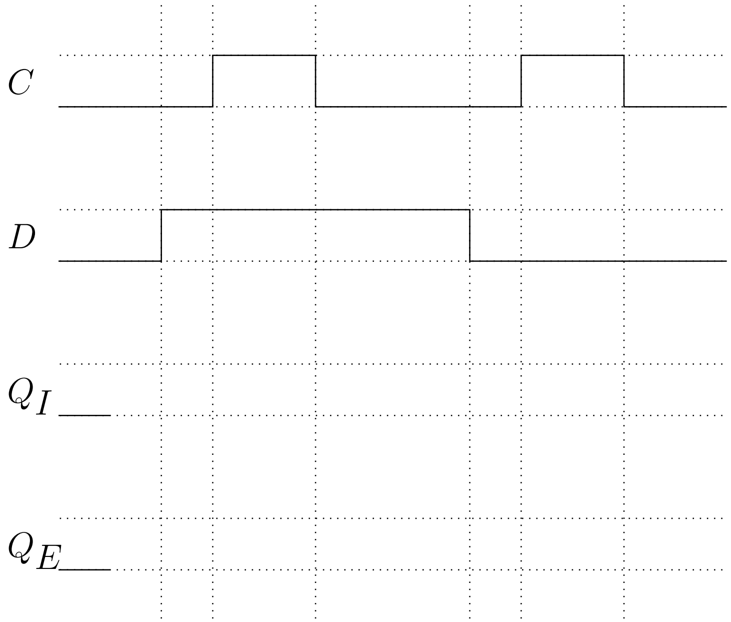


The D Flip-Flop

- We want state to be affected only at discrete points in time; a master-slave design achieves this.

Copyright figure removed

- Graphical example:

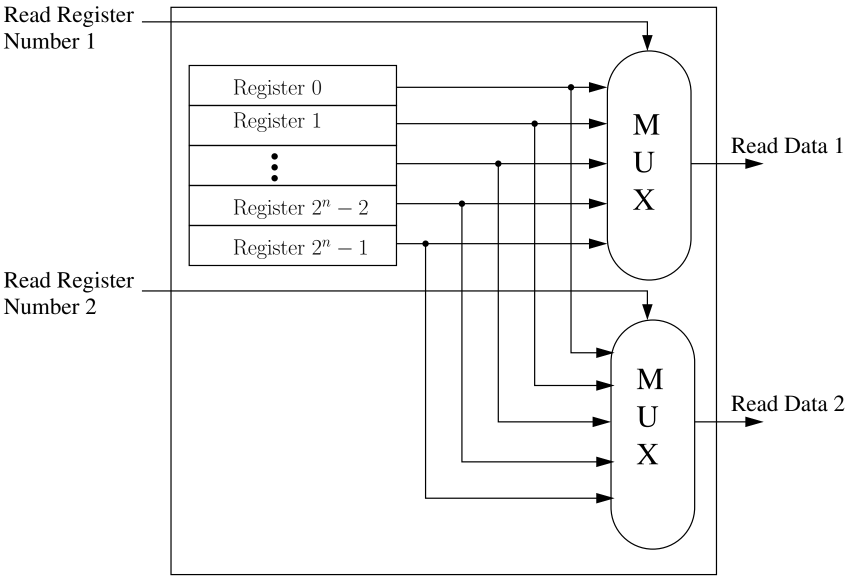


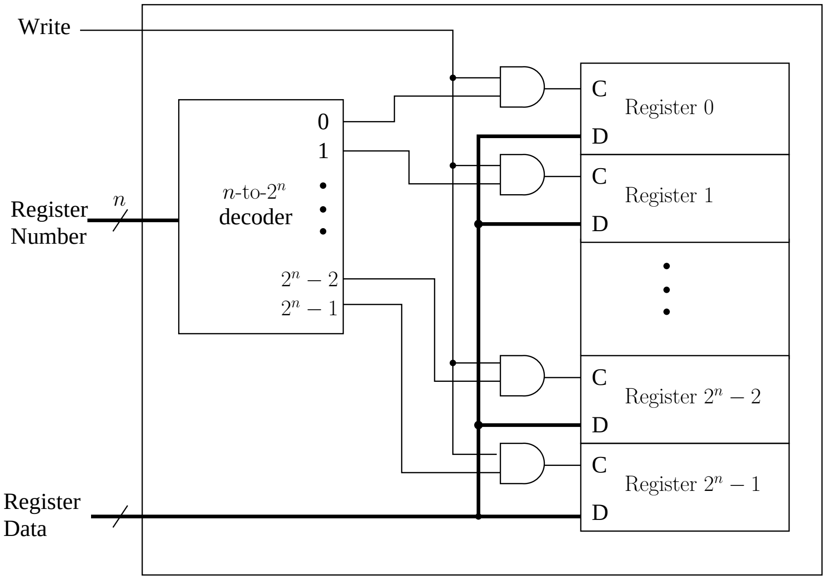
Registers and Register Files

- Register: an array of flip-flops (e.g. 32 for a word register)
- Register file: a way of organizing registers

Copyright figure removed

Read/Write Logic for Register File





Random Access Memories

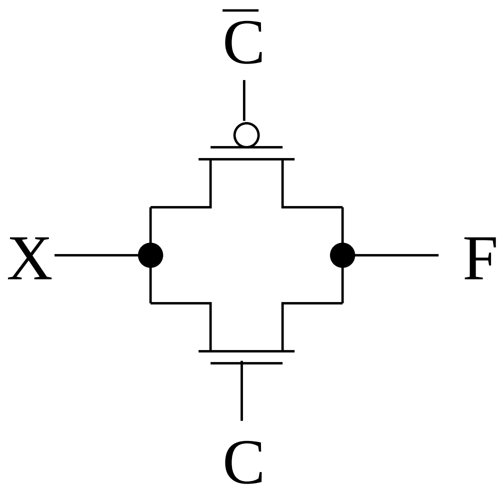
- Static random access memories (SRAM) use D latches

Copyright figure removed

- Register file idea won't scale up; decoder and multiplexors too big
- Fix multiplexor problem by using three-state buffers
- Fix decoder problem by using two-level decoding
- This type of memory is **not** clocked

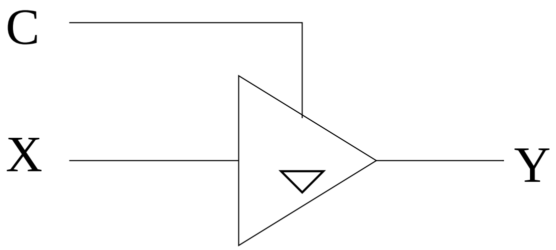
Three-state buffer or transmission gate

- Has three outputs 0, 1, and *floating* (connected to neither power or ground)

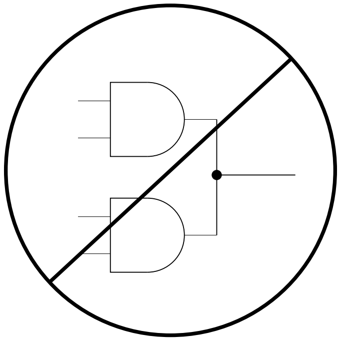


- $C = 1$, then
  - NMOS gate passes 0 well
  - $\bar{C} = 0$ and PMOS gate passes 1 well
- $C = 0$, then $\bar{C} = 1$ and both transistors are off (output is floating).

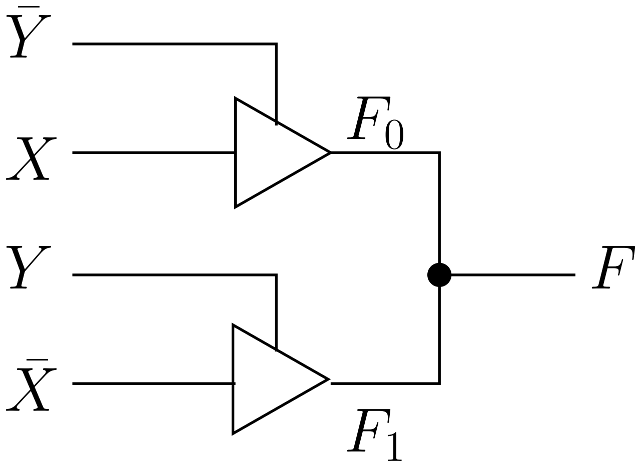
Using Three-State Buffers



- High-impedance outputs can be “tied together” without problems
- Normally, do not tie output lines together



XOR from Three-State Buffers



Circuit analysis with transmission gates:

- Label floating output as '—'
- Tied lines better have exactly one non-floating!

X	Y	$\bar{X}$	$\bar{Y}$	F_0	F_1	F
0	0					
0	1					
1	0					
1	1					

Making Multiplexors from Three-State Buffers

Copyright figure removed

IMPORTANT: Must ensure that at most one select input is 1, or short-circuit may result (physical meltdown)

Example of SRAM Structure

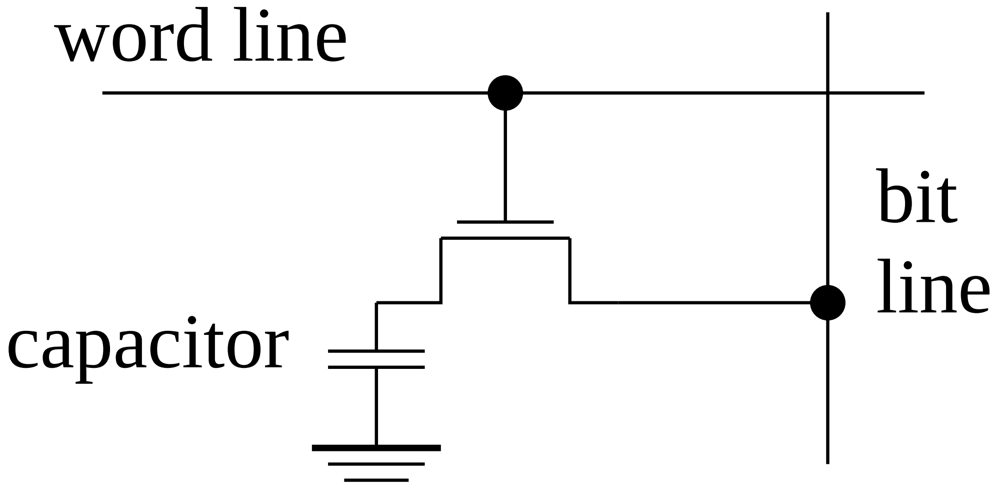
Copyright figure removed

Does this design scale up well?

Dynamic RAM

- Our SRAM cell uses a lot of transistors
- A better implementation uses six transistors
- This is still too expensive
- Alternative: use a capacitor to store a charge to represent 1
- Problem: charge leaks away, must be refreshed

DRAM Cell



- To write: place value on bit line
- To read: put half-voltage on bit line, 1 on word line
- Charge in capacitor will slightly increase bit line voltage, no charge will slightly decrease voltage
- This is detected, amplified, and written back

Design of 4Mx1 DRAM

Copyright figure removed

- 20-bit address provided 10 bits at a time
- Whole row is read at once
- Column address selects single bit
- Refresh handled a row at a time (external controller)
- If capacitors hold charge for 4ms, refresh takes 80ns, fraction of time devoted to refresh is about 4%

DRAM Complications

- DRAM is cheaper than SRAM, but slower
- Refresh controller must also allow read/write access
- Possibility of getting more bits out at a time (e.g. page-mode RAM)
- SDRAM: synchronized DRAM
  - Uses external clock to synchronize with processor
  - Useful in memory hierarchies

Designing Using Finite-State Machines

Copyright figure removed

High-level circuit implementation of finite-state machine

Example: Traffic Light

- Output signals: NSlight, EWlight
- Input signals: NScar, EWcar
- State names: NSgreen, EWgreen (no yellow for now)
- Functionality: want light to change only if car is waiting at red light

Graphical Representation of Traffic Light Controller

Copyright figure removed

- Names of states outside ovals
- Output in given state inside oval
- Transition arc labelled with Boolean formula of inputs

Variations on Finite-State Machines

- Moore machine: output depends only on state (what we use)
- Mealy machine: output can depend on inputs
- Moore machine may be faster, Mealy machine may be smaller
- Conceptually, computation is infinite (input streams have no beginning or end)
- In practice, need to worry about power-up and power-down (as with all our state devices)
- Different in language-recognition context (e.g. CS 241)
  - Input is single character at a time, not set of bits
  - Because strings have finite length, computation is finite (start state, final states)
  - Mealy machines used (outputs on transition arcs)

Electronic Implementation of Finite-State Controller

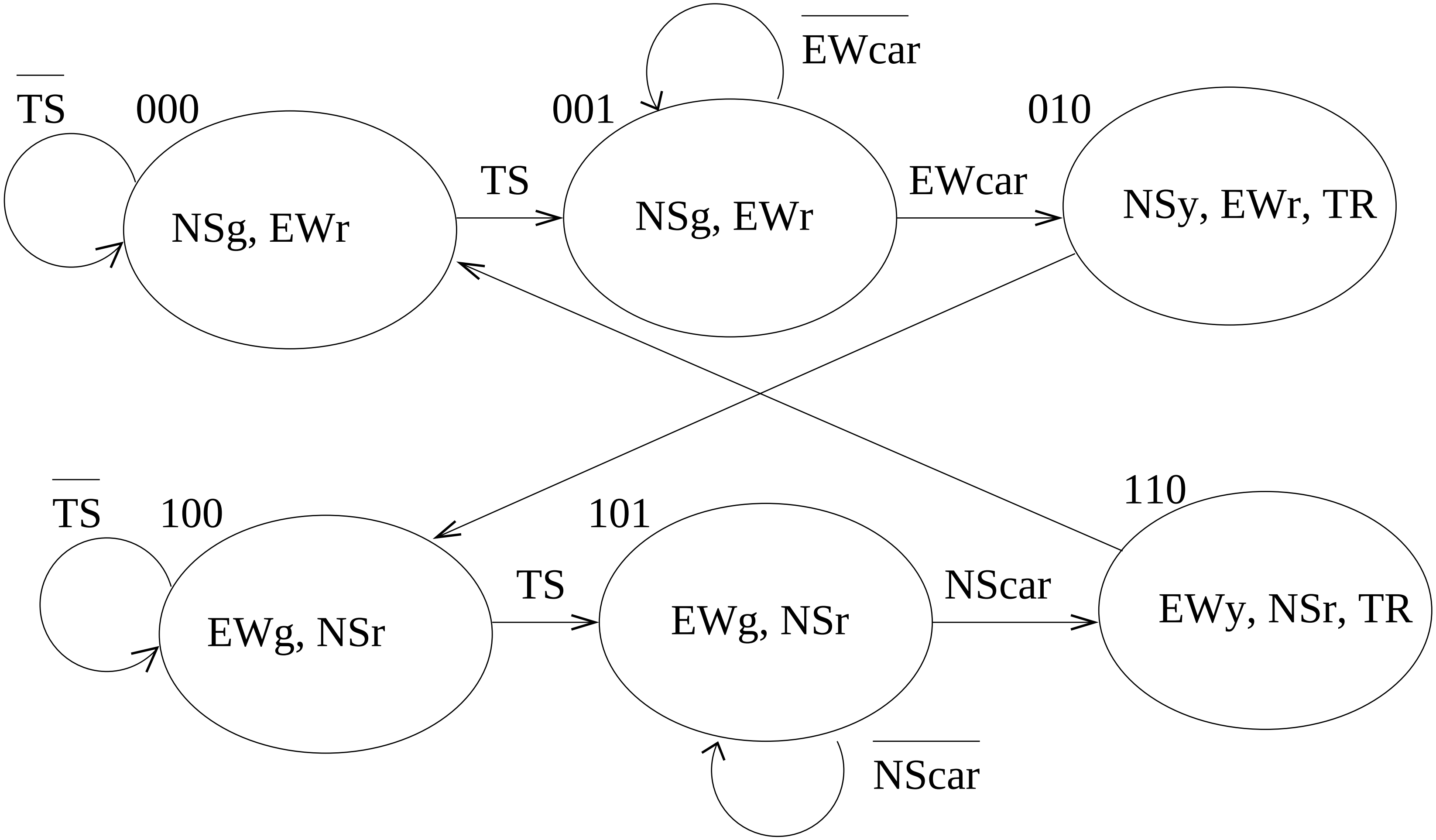
Copyright figure removed

Extending the Traffic-Light Controller

- Add 4-second yellow light
- Assume 0.25Hz clock
- need to add 28-second timer
  - Timer input: TimerReset (TR)
  - Timer output: TimerSignal (TS)
- Behaviour of system
  - Stay green in one direction (red in other direction) until car arrives or 32 seconds elapse, whichever happens last
  - Green turns to yellow for 4 seconds; red in other direction stays
  - Yellow turns to red, red in other direction turns to green

State Diagram of Extended Controller

- Inputs: NScar, EWcar, TS
- Outputs: NSg, NSy, NSr, EWg, EWy, EWr, TR



Next-State Table for Extended Controller

current	inputs			next	current	inputs			next
state	NS-	EW-		state	state	NS-	EW-		state
$S_2S_1S_0$	car	car	TS	$S'_2S'_1S'_0$	$S_2S_1S_0$	car	car	TS	$S'_2S'_1S'_0$
0 0 0	X	X	0	0 0 0	1 0 0	X	X	0	1 0 0
0 0 0	X	X	1	0 0 1	1 0 0	X	X	1	1 0 1
0 0 1	X	0	X	0 0 1	1 0 1	0	X	X	1 0 1
0 0 1	X	1	X	0 1 0	1 0 1	1	X	X	1 1 0
0 1 0	X	X	X	1 0 0	1 1 0	X	X	X	0 0 0
0 1 1	X	X	X	X X X	1 1 1	X	X	X	X X X

Note unused states, symmetries

Output Table For Extended Controller

- Output table looks like truth table
  Inputs are State, Outputs are Outputs
- Traffic light outputs: NSg, NSy, NSr, EWg, EWy, EWr, TR
- If output listed in State, then 1 in output table
  If output not listed in State, then 0 in output table

S_2	S_1	S_0	NSg	NSy	NSr	EWg	EWy	EWr	TR
0	0	0							
0	0	1							
0	1	0							
0	1	1							
1	0	0							
1	0	1							
1	1	0							
1	1	1							

Next-State/Output Logic For Extended Controller

Current state = $S_2S_1S_0$, next state = $S_2'S_1'S_0'$

$$S_0' = \overline{S_1}\,\overline{S_0} \cdot TS + \overline{S_2}\,\overline{S_1}\,S_0 \cdot \overline{EWcar} + S_2\overline{S_1}\,S_0 \cdot \overline{NScar}$$

$$S_1' = \overline{S_2}\,\overline{S_1}\,S_0 \cdot EWcar + S_2\overline{S_1}\,S_0 \cdot NScar$$

$$S_2' = \overline{S_2}\,S_1\overline{S_0} + S_2\overline{S_1}$$

$$NSg = \overline{S_2}\,\overline{S_1}, EWg = S_2\overline{S_1}$$

$$NSy = \overline{S_2}\,S_1\overline{S_0}, EWy = S_2S_1\overline{S_0}$$

$$NSr = S_2, EWr = \overline{S_2}$$

$$TR = S_1\overline{S_0}$$

Data Representation and Manipulation

- Readings from text:
  - 2.4, 3.1, 3.2
  - 2.6
  - B.5 **required reading**
  - 3.3
  - 3.5
  - ignore MIPS instructions for now
- How characters and numbers are represented in a typical computer
- Hardware designs which implement arithmetic operations

The MIPS Word

- 32-bit architecture
- 1 byte = 8 bits; 4 bytes = 1 word
- Bits numbered 31, 30, …, 0
- Most significant bit (MSB) is bit 31
- Least significant bit (LSB) is bit 0
- In many examples, we will use only 4 bits to illustrate
- Sometimes, numbers written in *hexadecimal*

Characters

- ASCII (American Standard Code for Information Interchange)
- Uses 7 bits to represent 128 different characters
- 8th bit (topmost) used as parity check (error detection)
- 4 characters fit into MIPS 32-bit word

  128 possibilities include upper and lower case Roman letters, punctuation marks, some computer control characters
- Partial table on page 106 of text
- Unicode: 16 bits per character

  (English isn't the only language!)

Unsigned Binary Numbers

- With 4 bits, can represent 0 through 15

$$\begin{aligned} 1101_2 &= (1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) \\ &= 13_{10} \end{aligned}$$

- With 32 bits, can represent 0 through $2^{32} - 1 = 4,294,967,295$
- How to represent negative numbers?

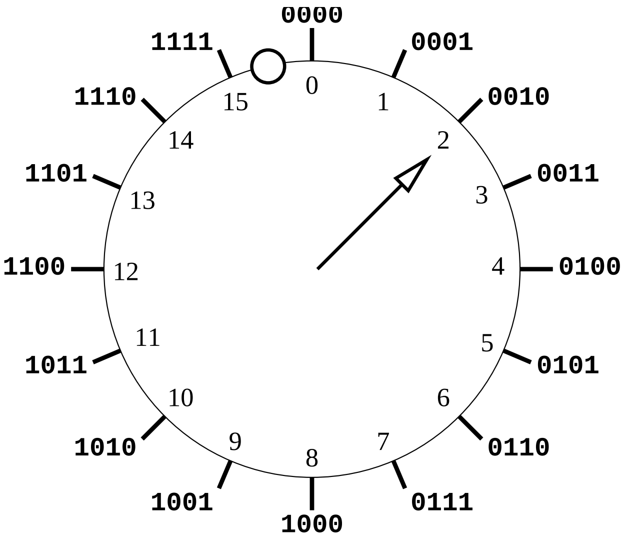
Signed Binary Numbers

- First idea: use MSB as "sign bit" (0 means positive, 1 means negative)
- Called "signed-magnitude" representation
- 4-bit example: 1110 is -6
- With 4 bits, can represent -7 (1111) to $+7$ (0111)
- Problems: two different versions of zero (0000 and 1000), addition is complicated
- Better idea: two's complement representation

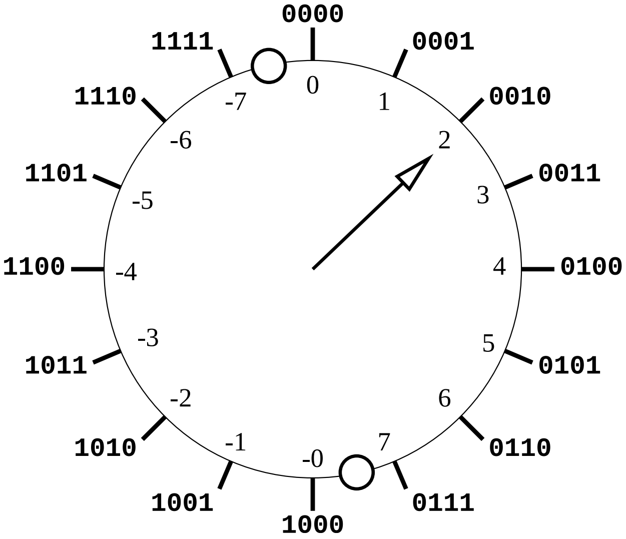
Two's Complement Representation

- Idea: Let MSB represent the negative of a power of 2
- With 4 bits, bit 3 (MSB) represents -2^3
- $1110 = -2^3 + 2^2 + 2^1 = -2$
- With 4 bits, can represent -8 (1000) to $+7$ (0111)
- With 32 bits, can represent $-2,147,483,648$ to 2,147,483,647
- Usefulness becomes apparent when we try arithmetic

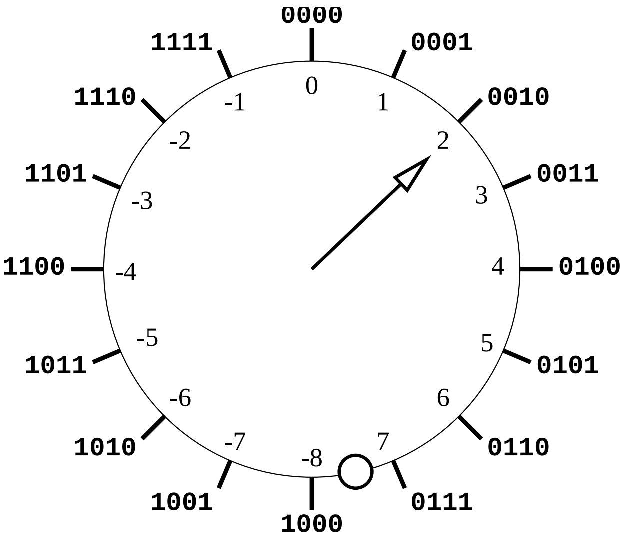
Pictorial Representation, 4 Bits



Unsigned



Signed Magnitude



Two’s Complement

Negating a Two's Complement Number

- For a bit pattern x, let $\bar{x}$ be the result of inverting each bit
- Example: $x = 0110, \bar{x} = 1001$
- Since $x + \bar{x} = -1, -x = \bar{x} + 1$
- To negate a number in two's complement representation, invert every bit and add 1 to the result

Sign Extension

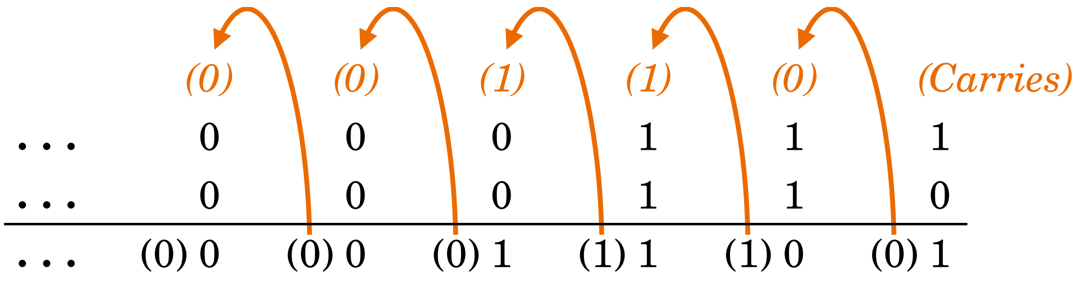
- With 4 bits, 0110 is +6. With 8 bits, what is +6?
- With 4 bits, 1010 is -6. With 8 bits, what is -6?
- To expand number of bits used, copy old MSB into new bit positions.
- This works because

$$-2^i + 2^{i-1} + 2^{i-2} + \cdots + 2^{j+1} + 2 \cdot 2^j = 0$$

Addition

- To add two two's complement numbers, simply use the "elementary school algorithm", throwing away any carry out of the MSB position
- To subtract, simply negate and add
- Problem: what if answer cannot be represented? (called overflow)
- Overflow in addition cannot occur if one number is positive and the other negative
- If both addends have same sign but answer has different sign, overflow has occurred

Building An Addition Circuit



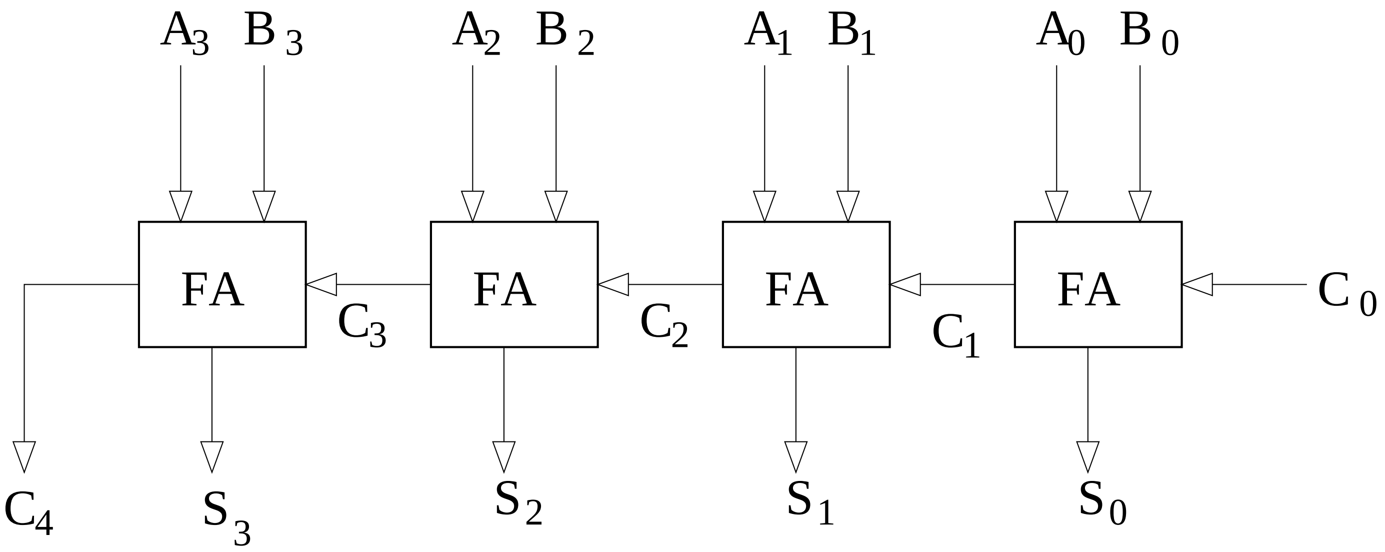
	(0)	*(0)*	*(1)*	*(1)*	*(0)*	*(Carries)*
. . .	0	0	0	1	1	1
. . .	0	0	0	1	1	0
. . .	(0) 0	(0) 0	(0) 1	(1) 1	(1) 0	(0) 1

Basic building block: Full Adder (takes three bits as input, outputs 2-bit sum)

Copyright figure removed

Ripple-Carry Adder

- 4-bit example



- Easy to extend to 32 bits
- Can be slow; “carry-lookahead” idea improves speed

Logical Operations

- shifting bits in word left or right (shifting in 0, called “logical shift”)
- shift right, duplicating old MSB (called “arithmetic right shift”)
- rotate left or right (moving bit rotated out to other side)
- bitwise AND, OR, NOR (bitwise NOT is just NOR with all-zero word)

A 1-Bit ALU

Copyright figure removed

- Extends functionality of full adder
- Performs AND, OR, addition
- Connect 32 of these as with ripple-carry adder to perform 32-bit operations

Improving the 1-Bit ALU

- How to implement subtraction?
- To subtract *b* from *a*, invert bits of *b*, add to *a*, add 1
- Box below will do this, if added 1 is put into CarryIn at top of chain when subtraction is desired

Copyright figure removed

Abstracting Away ALU Details

- Book makes further improvements to support other operations that assist in branching (Appendix B.5—**required reading**; figures B.5.9 and B.5.10 in particular)
- From now on, we use symbol below
- Same shape used for ripple-carry adder, so remember to label them

Copyright figure removed

Multiplication

- Example:

$$
\begin{array}{r c cccccccc l}
13 & & & & & & 1 & 1 & 0 & 1 & \text{Multiplicand} \\
11 & & & & & & 1 & 0 & 1 & 1 & \text{Multiplier} \\
\cline{1-1} \cline{7-10}
 & & & & & & 1 & 1 & 0 & 1 & \\
 & & & & & 1 & 1 & 0 & 1 & & \\
 & & & & 0 & 0 & 0 & 0 & & & \\
 & & & 1 & 1 & 0 & 1 & & & & \\
\cline{1-1} \cline{3-10}
143 & & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & \text{Product}
\end{array}
$$

- Algorithm for Product = Multiplicand * Multiplier:

  Look at LSB of Multiplier

  If 1, add Multiplicand to Product

  Shift Multiplicand left, Shift Multiplier right

  Repeat until Multiplier becomes zero

- Note: for *n* bit numbers, result may be 2*n* bits

Multiplication Hardware, First Version

Copyright figure removed

- Initialization and termination not shown
- At start, Product is zero, 32-bit Multiplicand in right half of its register
- Note control inputs and outputs

Copyright figure removed

4-Bit Multiplication Example, First HW Version

Multiplier = 1011, Multiplicand = 1101

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial Values	1011	0000 1101	0000 0000
1	Add mpcd to prod			0000 1101
	Shift left mpcd		0001 1010	
	Shift right mplr	0101		
2	Add mpcd to prod			0010 0111
	Shift left mpcd		0011 0100	
	Shift right mplr	0010		
3	No operation			
	Shift left mpcd		0110 1000	
	Shift right mplr	0001		
4	Add mpcd to prod			1000 1111
	Shift left mpcd		1101 0000	
	Shift right mplr	0000		

Multiplication Hardware, Second Version

Copyright figure removed

- Multiplicand added to left half of product register
- As low-order bits of product become fixed, they are shifted into right half of product register

Copyright figure removed

4-Bit Multiplication Example, Second HW Version

Multiplier = 1011, Multiplicand = 1101

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial Values	1011	1101	0000 0000
1	Add mpcd to prod			1101 0000
	Shift right prod		1101	0110 1000
	Shift right mplr	0101		
2	Add mpcd to prod			$\boxed{1}$0011 1000
	Shift right prod		1101	1001 1100
	Shift right mplr	0010		
3	No operation			
	Shift right prod		1101	0100 1110
	Shift right mplr	0001		
4	Add mpcd to prod			$\boxed{1}$0001 1110
	Shift right prod		1101	1000 1111
	Shift right mplr	0000		

Multiplication Hardware, Third Version

Copyright figure removed

- Multiplier starts in right half of product register
- As multiplier bits are shifted out, unchanging bits of product are shifted into the space created

Copyright figure removed

4-Bit Multiplication Example, Third HW Version

Multiplier = 1011, Multiplicand = 1101

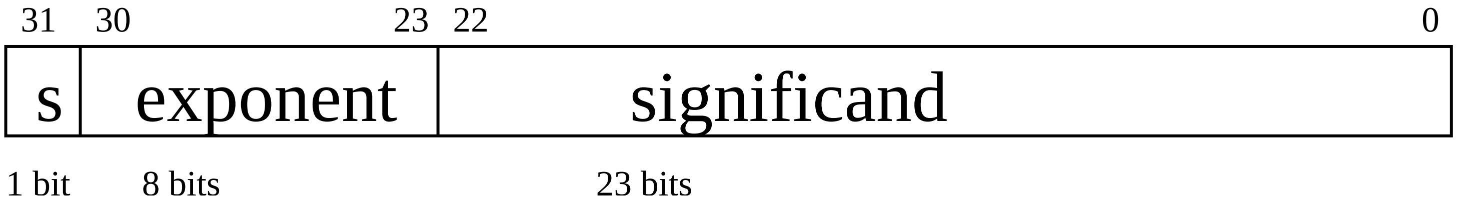
Iteration	Step	Multiplicand	Product
0	Initial Values	1101	$0000\ \boxed{1011}$
1	Add mpcd to prod		$1101\ \boxed{1011}$
	Shift right prod	1101	$0110\ 1\boxed{101}$
2	Add mpcd to prod		$\boxed{1}0011\ 1\boxed{101}$
	Shift right prod	1101	$1001\ 11\boxed{10}$
3	No operation		
	Shift right prod	1101	$0100\ 111\boxed{1}$
4	Add mpcd to prod		$\boxed{1}0001\ 111\boxed{1}$
	Shift right prod	1101	1000 1111

Representing Numbers That Aren't Integers

- Uses idea of scientific notation: -3.45×10^3
- Sign, significand (fraction, mantissa, exponent)
- Normalized: single digit to left of decimal point
- For computers, natural to use 2 as base
- Example: $1.01_2 \times 2^4$
- In normalized binary, leading digit of significand is always 1 (can omit it from internal representation)
- How to represent 0?

Floating-Point Representation

- MIPS uses the IEEE 754 floating-point standard format



31 30 23 22 0

s | exponent | significand

1 bit 8 bits 23 bits

- allows numbers from 2.0×10^{-38} to 2.0×10^{38}, roughly
- Double precision: uses two 32-bit words, 11 bits for exponent, 52 bits for significand
- Exponent is stored in “biased” notation: most negative exponent is all 0’s, most positive is all 1’s

  This allows for quick comparisons, speeds up sorting
- Thus value represented is

  $(-1)^{S} \times (1 + \text{Significand}) \times 2^{(\text{Exponent} - \text{Bias})}$,

  where Bias $=$ 127 for single precision
- Special case: 0000 0000 exponent reserved for 0

Fractional Numbers

- How to represent numbers less than 1?
- Digits to right of decimal point represent negative powers of two

0.	1	0	1	1
1	1/2	1/4	1/8	1/16

0*1 + 1*1/2 + 0*1/4 + 1*1/8 + 1*1/16 = 11/16

- Simple examples

1/2 = 0.1

3/4 = 0.11

- May have to approximate

Example: 1/3 as decimal is...

0.1 in binary is...

sqrt(2) in binary is...

Floating-Point Addition

- Decimal example: $9.54 \times 10^2 + 6.83 \times 10^1$

  (assume we can only store two digits to right of decimal point)

  1. Match exponents: $9.54 \times 10^2 + .683 \times 10^2$
  2. Add significands, with sign: 10.223×10^2
  3. Normalize: 1.0223×10^3
  4. Check for exponent overflow/underflow
  5. Round: 1.02×10^3
  6. May have to normalize again

- Same idea works for binary

Copyright figure removed

Copyright figure removed

Floating-Point Multiplication

- Decimal example: $(9.54 \times 10^2) \times (6.83 \times 10^1)$
  (assume we can only store two digits to right of decimal point)
  1. Add exponents: $2 + 1 = 3$
     (Note: exponents stored in biased notation)
  2. Multiply significands: $9.54 \times 6.83 = 65.1582$
  3. Unnormalized result: 65.1582×10^3
  4. Normalize: 6.51582×10^4
  5. Check for overflow/underflow
  6. Round: 6.52×10^4
     (May need to renormalize)
  7. Set sign
- Same idea works for binary

Accuracy

- Only certain numbers can be represented accurately
- Typically the result of an operation cannot be represented precisely
- The result must be rounded. Multiple ways to round

  For this class, we will round 1/2 up
- Do we need to compute precisely and then round?
- Goal: save hardware by not keeping full precision internally during computation
- How few bits can be used to get correct *n*-bit result after rounding?

  Result should be the same as if we had kept full precision and rounded afterwards

Accuracy in Floating-Point Addition

- In adding two significands with n bits of precision, can use an n-bit adder (giving $n+1$-bit result)
- Is least significant bit of result enough to round correctly?
- Our addition examples will use $n = 4$

$$\begin{array}{ccccccccc} & & 1 & . & 0 & 1 & 0 & \times & 2^0 \\ + & & 1 & . & 1 & 1 & 1 & \times & 2^0 \\ \hline & 1 & 1 & . & 0 & 0 & 1 & \times & 2^0 \end{array}$$

Hardware adder gives two bits to left of decimal point

- This is normalized to 1.1001×2^1 and then rounded to 1.101×2^1
- Problem may arise when one significand has to be shifted to match exponents

- Example: $1.010 \times 2^2 + 1.001 \times 2^1$

  After normalization, our input bits span range of $n+1$ bits.

  How do we add this with an n-bit adder?

  Can we ignore low order bit?

$$\begin{array}{cccccccccc} & & 1 & . & 0 & 1 & 0 & & \times & 2^2 \\ + & & 0 & . & 1 & 0 & 0 & \boxed{1} & \times & 2^2 \\ \hline & 0 & 1 & . & 1 & 1 & 0 & & \times & 2^2 \end{array}$$

  Note leftmost 0 is carry out of adder

- Here, boxed bit of second significand was not fed into adder

  But is boxed bit needed to round correctly?

- With it, normalized result is 1.1101×2^2, rounds to 1.111×2^2
- Without it, normalized result is 1.110×2^2
- Thus for n-bit accuracy, we need to keep $n+2$ bits during the computation

Accuracy in Floating-Point Multiplication

- When multiplying two floating-point numbers, the significands are multiplied together
- If the significands have n bits of precision each, the result can have $2n$ bits of precision
- How many bits do we need to keep during the computation?
- Our multiplication examples will have $n = 3$

- Example:

$$\begin{array}{rcccccccl} & & 1 & . & 1 & 1 & & & \times\ 2^2 \\ \times & & 1 & . & 1 & 1 & & & \times\ 2^1 \\ \hline & 1 & 1 & . & 0 & 0 & 0 & 1 & \times\ 2^3 \end{array}$$

- In above example, only top 3 bits are needed for final result of 1.10×2^4
- Example: Do we need circled (fourth) bit?

$$\begin{array}{rcccccccl} & & 1 & . & 1 & 0 & & & \times\ 2^2 \\ \times & & 1 & . & 1 & 0 & & & \times\ 2^1 \\ \hline & 1 & 0 & . & 0 & \boxed{1} & 0 & 0 & \times\ 2^3 \end{array}$$

- With three bits, 10.0×2^3 is normalized to 1.00×2^4, which is incorrectly rounded
- With four bits, 10.01×2^3 is normalized to 1.001×2^4, and correctly rounded up to 1.01×2^4

- Example: Do we need circled (fourth,fifth) bits?

$$\begin{array}{ccccccccl}
 & & 1 & . & 0 & 1 & & & \times\ 2^2 \\
\times & & 1 & . & 1 & 0 & & & \times\ 2^1 \\
\hline
 & 0 & 1 & . & 1 & \boxed{1} & \boxed{1} & 0 & \times\ 2^3
\end{array}$$

- With three bits, 01.1×2^3 is normalized to 1.10×2^3, which is incorrectly rounded
- With four bits, 01.11×2^3 is normalized to 1.110×2^3, and rounded to 1.11×2^3, which is incorrectly rounded
- With five bits, 01.111×2^3 is normalized to 1.111×2^3, rounded to 10.0×2^3, and normalized again to 1.00×2^4, which is correctly rounded

Floating-Point Architectural Issues

- To maintain n bits of accuracy after an operation, preserve $n+2$ bits during the computation (the two extra bits are sometimes called *guard* and *round*)
- Separate floating-point registers?
- Separate floating-point coprocessors?
- Rounding or truncating?
- What to do about overflow (same issue as for integer arithmetic)?

Single-Cycle Processor Implementation

- Readings from text: Chapter 4, sections 4.1–4.4; Appendix D, section D.2
- How to build datapath, control for specific architecture
- We will implement small subset of MIPS operations:
  - Load (`lw`) and store (`sw`)
  - Add (`add`), subtract (`sub`), AND (`and`), OR (`or`), set on less than (`slt`)
  - Branch-equal (`beq`) and jump (`j`)
- These suffice to illustrate fundamental ideas

Review of MIPS Architecture

- 32 registers (numbered 0 to 31), each with 32 bits
- Register 0 always supplies the value 0
- Memory of 32-bit words
- Memory is byte-addressable (word addresses multiples of 4)
- Words have the address of their most significant byte
- All MIPS instructions are 32 bits long

Review of MIPS Instructions

- Load: `lw $s1, 100($s2)`
  - Operands are register to be loaded, address in memory
  - Addressing modes: register, base (displacement), immediate
- Add: `add $s1, $s2, $s3`
  - Operands are destination register, two source registers
- Branch equal: `beq $s0, $s1, 10`
  - Operands are registers to be compared, relative jump offset
- Jump: `j 3000`
  - Operand is word address of next instruction
    (need to multiply by 4)

High-Level View of MIPS Functional Units

Copyright figure removed

- PC: Program Counter (address of current instruction)
- Fetch-execute cycle:
  - Fetch instruction (update PC)
  - Execute instruction
    - Fetch register operands
    - Compute result
    - Store into registers OR use to index memory

First Implementation: One Cycle Per Instruction

- Simpler to understand, but not practical
- Requires separate instruction and data memories
- Clock must be slowed to speed of slowest instruction
- Subsequently we look at multicycle implementations

Implementing Fetch Portion of Fetch-Execute

Copyright figure removed

- State elements here are PC (register) and instruction memory
- Adder is combinational

Datapath components for R-type instructions

- Example: `add $t1, $t2, $t3`

Copyright figure removed

- Note design permits read/write of same register

Datapath components for load/store instructions

- Example: `lw $t1, 100($t2)`

Copyright figure removed

- Sign extend is combinational
- Assume for simplicity that data memory is edge-triggered

Datapath components for branch instructions

- Example: `beq $t1, $t2, 100`

Copyright figure removed

- Shift is necessary because offset given is in words
- Still need mechanism to control PC loading

Assembled Single-Cycle Datapath

Copyright figure removed

- Note multiplexors added to “reuse” units

Instruction Formats

R-format: `add $t1, $t2, $t3` ⇒ `add rd,rs,rt`

31 26	25 21	20 16	15 11	10 6	5 0
0	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Load/store: `lw $t1, 100($t2)` ⇒ `lw rt,100(rs)`

31 26	25 21	20 16	15 0
35/43	rs	rt	offset
6 bits	5 bits	5 bits	16 bits

Jump: `j 3000`

31 26	25 0
4	address
6 bits	26 bits

- First field is operation code (opcode) but `add, sub,` etc. don’t need separate opcodes (funct)
- Note “destination register” field is different for `add` and `lw`; this complicates the datapath some more

Copyright figure removed

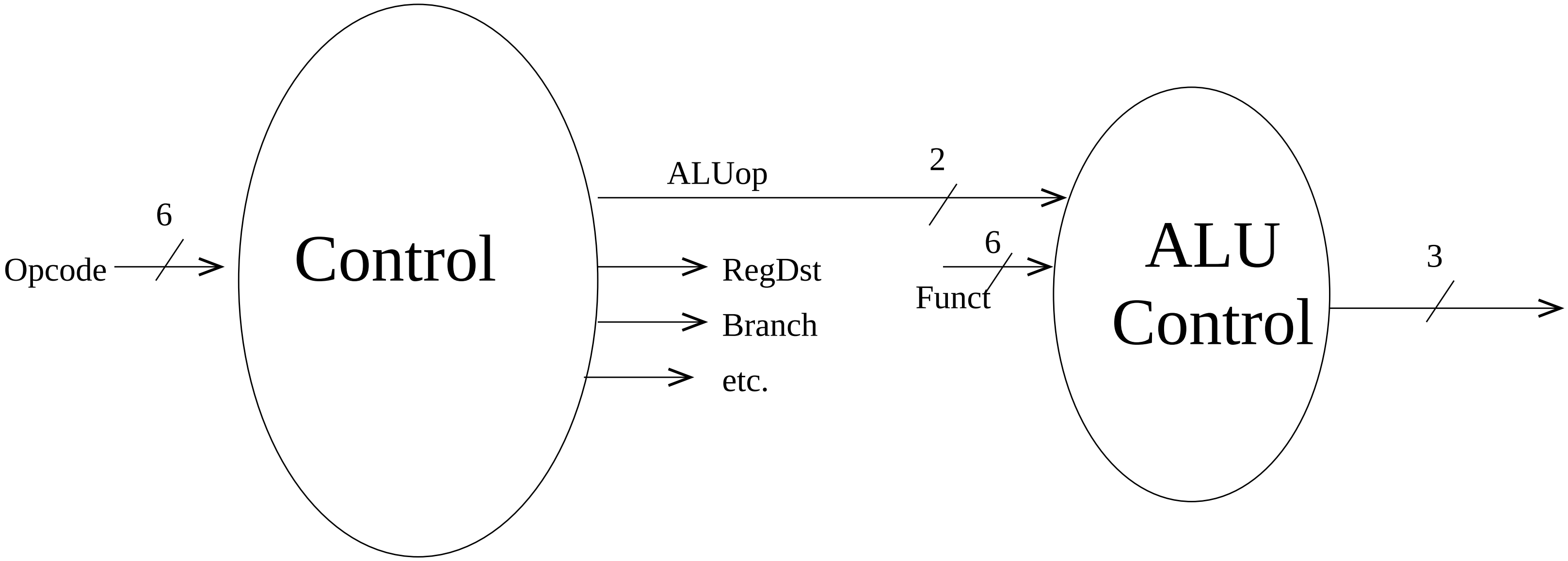
Meaning of Signals in Single-Cycle Datapath

Signal	Signal=0	Signal=1
RegDst	rt used	rd used
RegWrite	no effect	register written
ALUSrc	ALU B input from reg	immediate from instruction
Branch*	no branch	Branch
MemRead	no effect	memory read
MemWrite	no effect	memory written
MemToReg	reg write from ALU	reg write from memory

* Branch is ANDed with Zero from ALU to get PCsrc

(Full version in Figure 4.16 of text.)

Overview of Single-Cycle Control



- Could be done in one level
- Multiple levels of control are conceptually simpler
- Smaller control units may also be faster
- Readings: Appendix D, section D.2

Designing Single-Cycle Control

Mapping of operation to ALU control input:

Opcode	Operation	ALUop	Funct	ALU action	ALU ctrl input
35	`lw`	00	XXXXXX	add	0010
43	`sw`	00	XXXXXX	add	0010
4	`beq`	01	XXXXXX	subtract	0110
0	`add`	10	100000	add	0010
0	`sub`	10	100010	subtract	0110
0	`and`	10	100100	AND	0000
0	`or`	10	100101	OR	0001
0	`slt`	10	101010	slt	0111

Bottom five instructions use Funct bits to determine ALU action

Designing ALU control

- Truth table for ALU control bits, expanded

ALUop		Funct field						Operation
ALUop1	ALUop0	F5	F4	F3	F2	F1	F0	3210
0	0	X	X	X	X	X	X	0010
X(0)	1	X	X	X	X	X	X	0110
1	X(0)	X	X	0	0	0	0	0010
1	X(0)	X	X	0	0	1	0	0110
1	X(0)	X	X	0	1	0	0	0000
1	X(0)	X	X	0	1	0	1	0001
1	X(0)	X	X	1	0	1	0	0111

- Split ALU control input as Operation3,Operation2, Operation1, Operation0

  (Operation3=0 for our subset of MIPS)

Circuitry for ALU control

Copyright figure removed

Implementing Main Control Function

Type	Reg Dst	ALU Src	Mem ToReg	Reg Write	Mem Read	Mem Write	Branch	ALU op1	ALU op0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

Type	Dec Opcode	Binary Opcode
R-format	0	000 000
`lw`	35	100 011
`sw`	43	101 011
`beq`	4	000 100

Using A PLA To Implement Main Control

Copyright figure removed

- Method used for ALU control does not scale up well

Performance of Single Cycle Machines

- Suppose memory units take 200 ps (picoseconds), ALUs 100 ps, register files 50 ps, no delay on other units
- Jumps take 200 ps, branches take 350 ps, R-format instructions 400 ps, stores 550 ps, loads 600 ps.
- Clock period must be increased to 600 ps or more
- Even worse when floating-point instructions are implemented
- Idea: use multicycle implementation and R format

Modifying the datapath

- Normally design complete datapath for all instructions together.
- Various ways to modify datapath. The following is one approach for adding a new assembly instruction:

1. Determine what datapath is needed for new command
2. Check if any components in current datapath can be used
3. Wire in components of new datapath into existing datapath
   Probably requires MUXes
4. Add new control signals to Control units
5. Adjust old control signals to account for new command

- Add `jrel $ra` which performs
  PC ← PC + 4 + 4*$ra

Copyright figure removed

Multicycle Processor Implementations

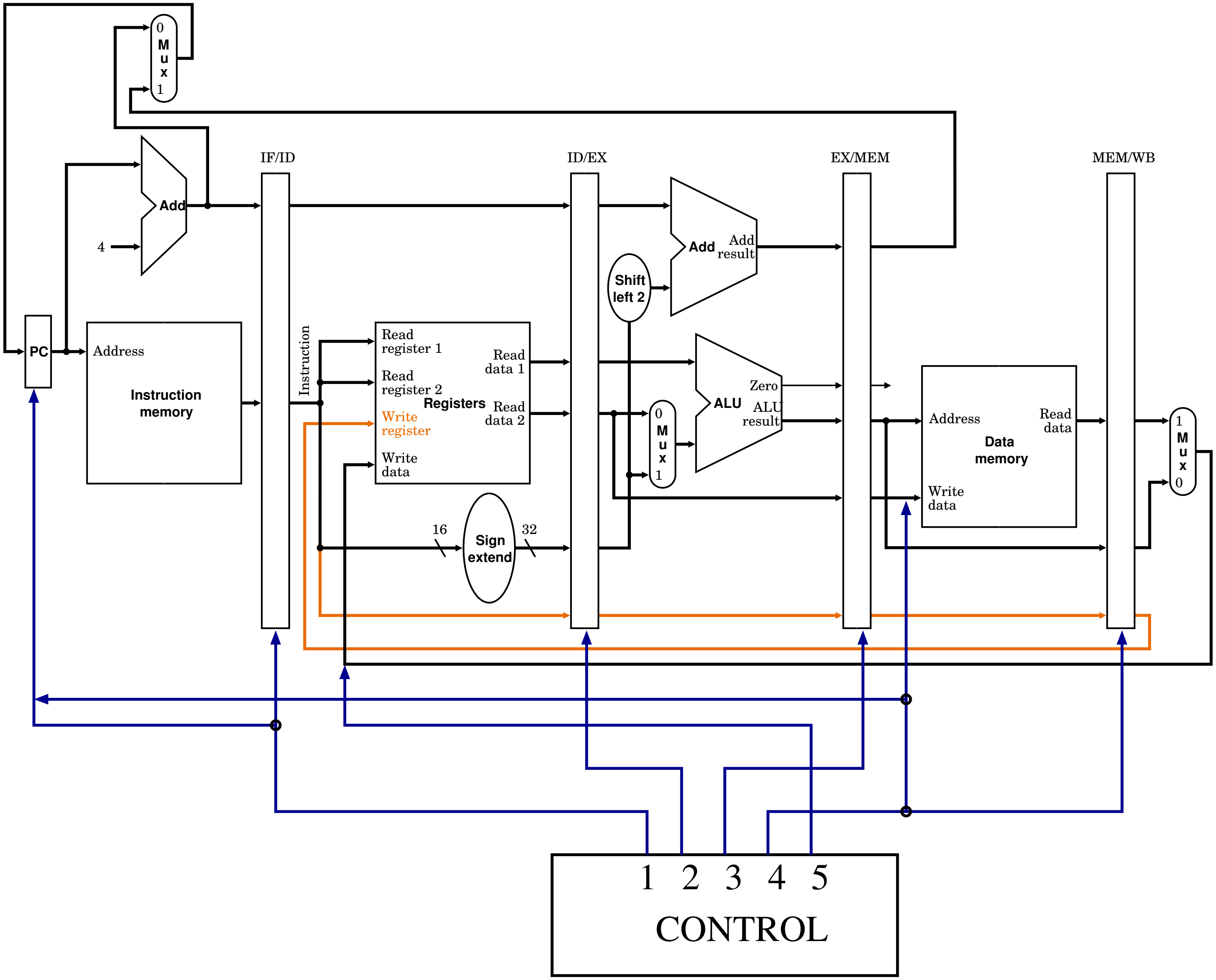
- NO READINGS
- Use several clock cycles to execute one instruction
- Single memory for intructions and data now possible
  Shown as separate memories on slides
- At end of clock cycle, all data used in subsequent cycles must be stored in state element
- We assume one clock cycle can contain one memory access, a register file access (two reads or one write), or one ALU operation

Datapath for Multicycle Implementation

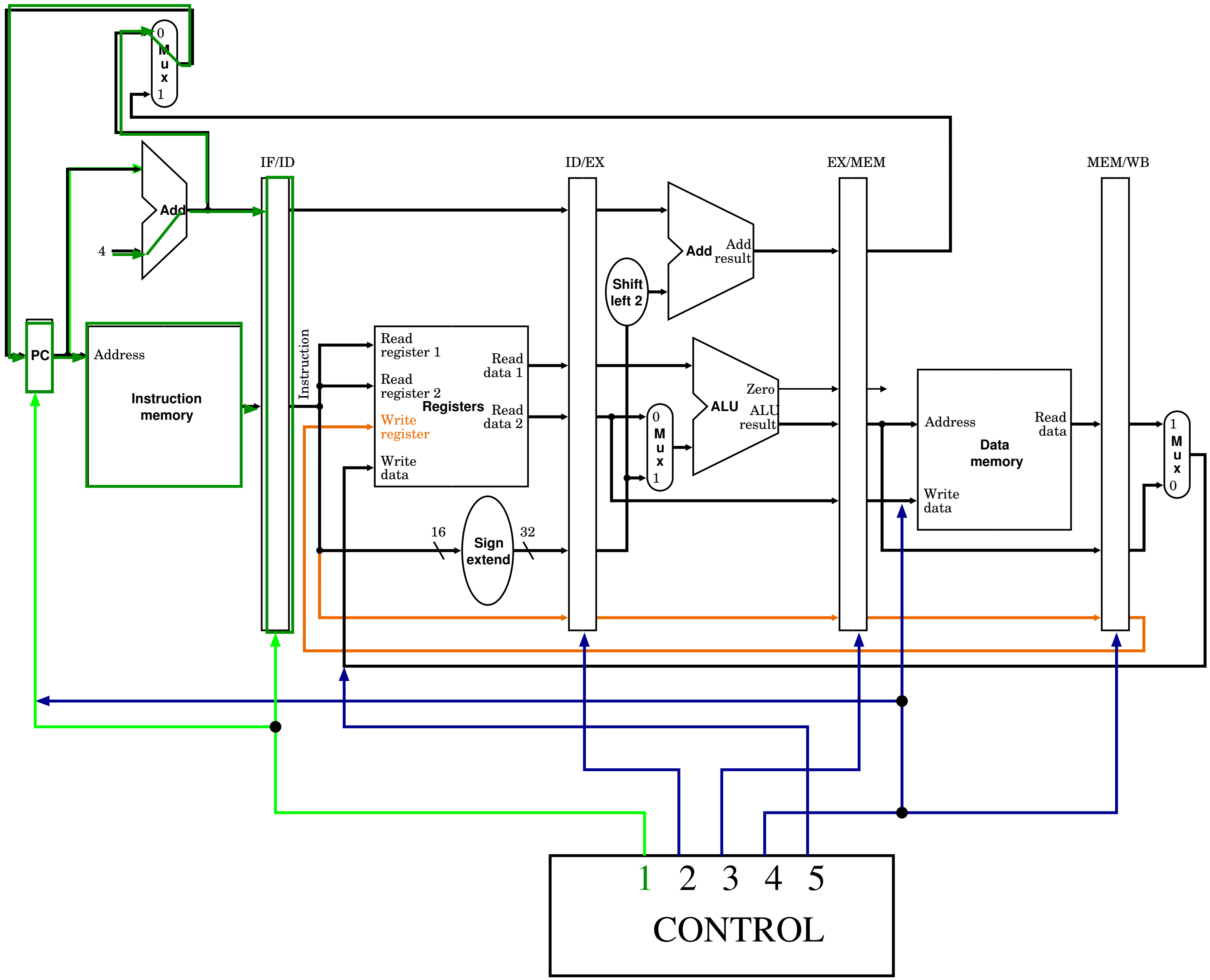
Copyright figure removed

- Register files (IF/ID, etc) store information computed in one stage that’s needed by later stages

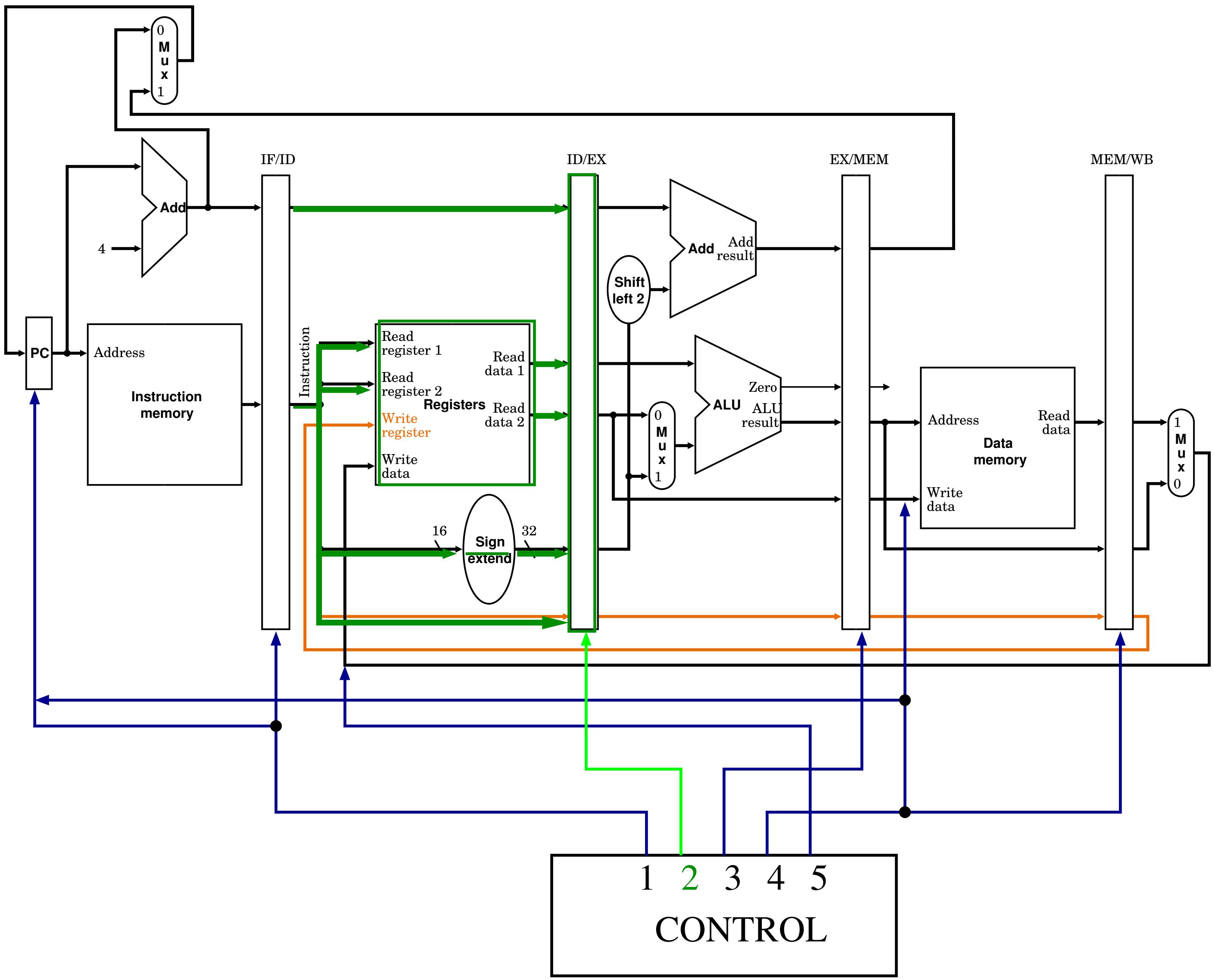
Datapath/Control for Multicycle Implementation



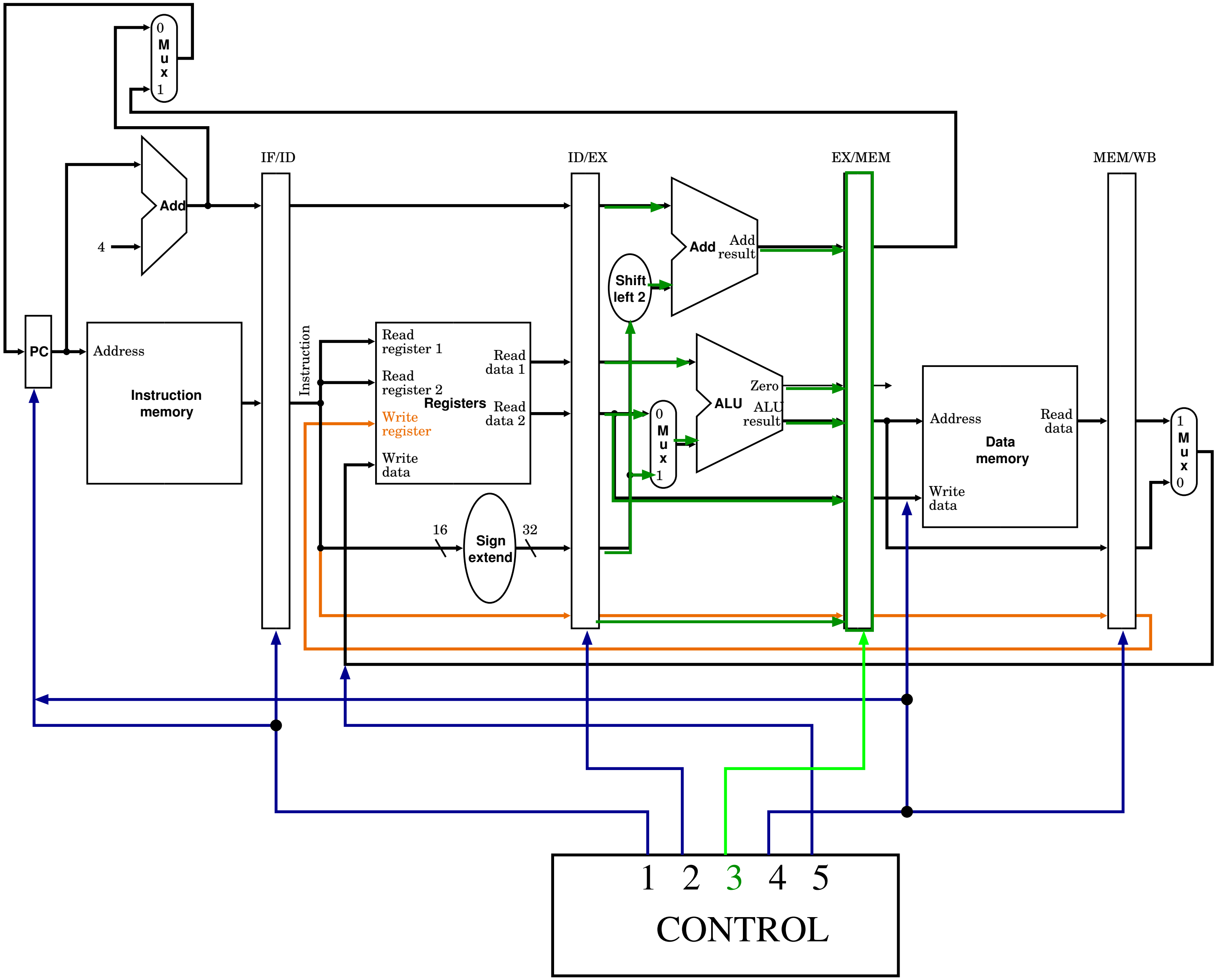
Instruction Fetch



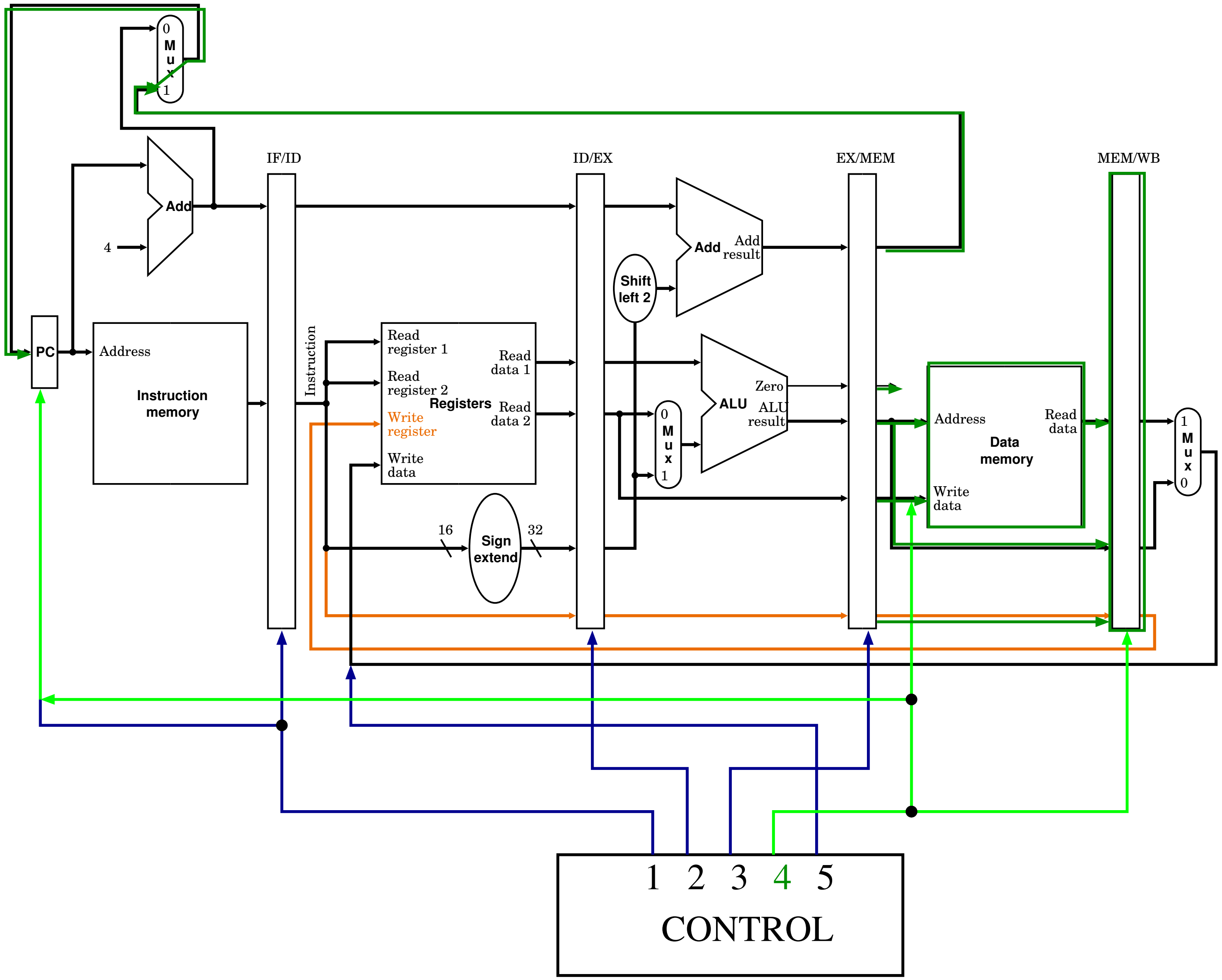
Instruction Decode



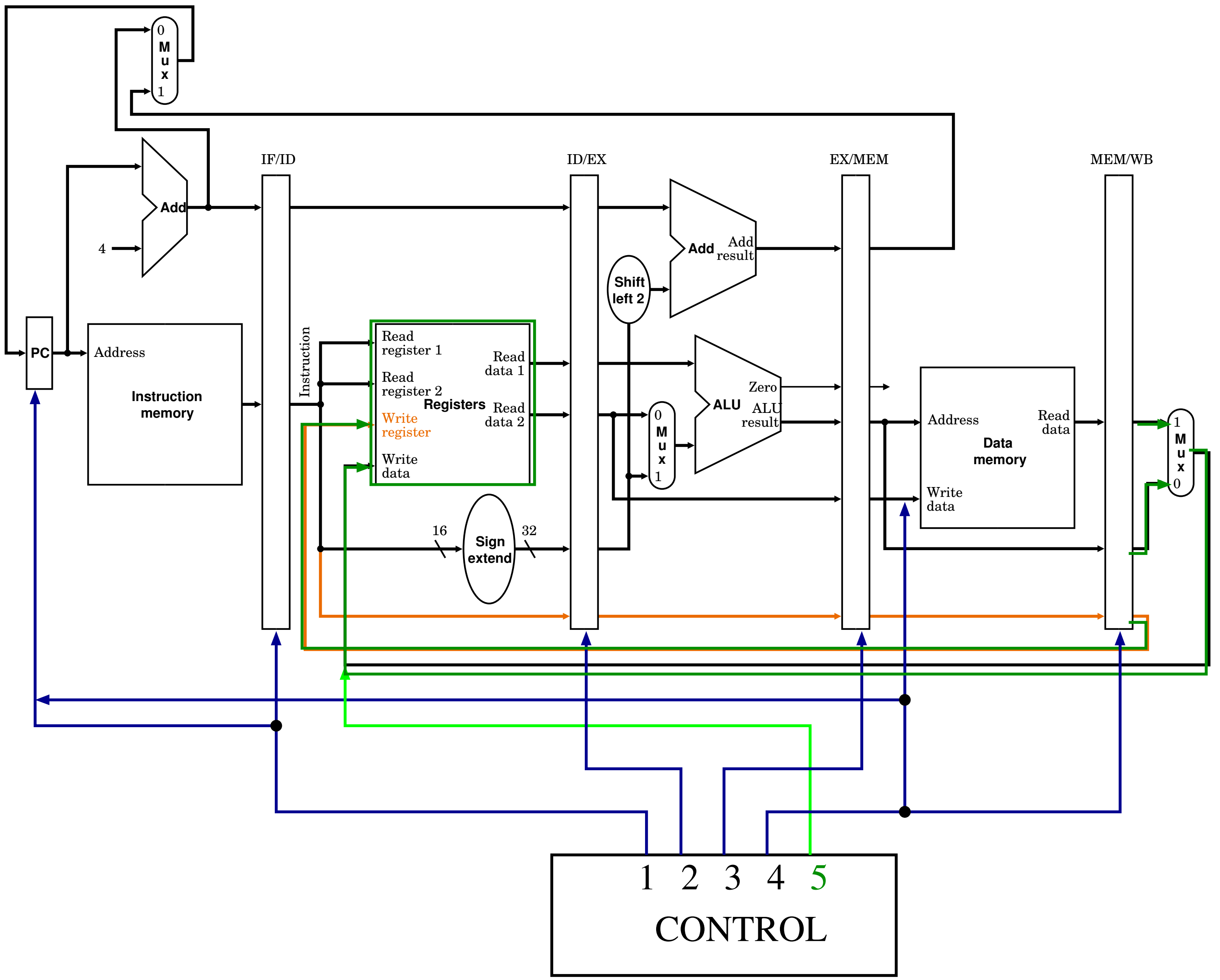
Instruction Execute



Memory (and...)


0
M
u
x
1
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
16
Sign extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
0
M
u
x
1
Address
Data memory
Read data
Write data
1
M
u
x
0
1 2 3 4 5
CONTROL

Write Back


0
M
u
x
1
Add
4
PC
Address
Instruction memory
IF/ID
Instruction
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
16
Sign extend
32
ID/EX
Shift left 2
Add
Add result
ALU
Zero
ALU result
0
M
u
x
1
EX/MEM
Address
Data memory
Write data
Read data
MEM/WB
1
M
u
x
0
1 2 3 4 5
CONTROL

Control

- Control steps through states 1,2,3,4,5 and repeats
- Can imagine smarter control:
  - `beq` and `sw` finish after state 4
  - `j` (not shown) finish after state 2

Performance

- Single cycle computer has 600ps clock
- Multicycle clock must be speed of *slowest* component. 200ps
- Each instruction takes 5 clock cycles

  Instructions use every stage, even some stages not needed
- Time per instruction: 5×200ps = 1000ps

  Much slower than single cycle!
- Smarter control: 4 cc for 800ps.

Notes on the Multicycle Datapath

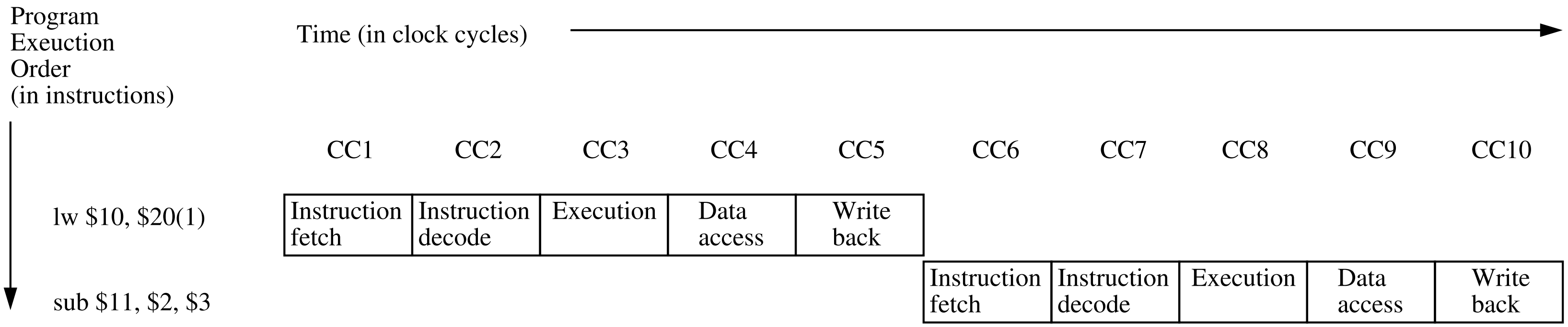
- Real multicycle computer:
  - Much more hardware (stages)
    Not all stages used for all instructions
    Instructions only use stages they need
  - More complex instruction set
    Instructions will take different times to execute
  - Fast instructions take less time on multicycle
    since single cycle executes at speed of slowest
  - Control far more complex than ours (internal loops, branches)
    Finite State Machine
    Microprogramming
  - At any time, about 80% of hardware is unused

Pipelining

- Readings: Chapter 4, sections 4.5–4.9
- Idea: increase parallelism by overlapping execution of multiple instructions
- Analogy: laundry (wash/dry/fold/put-away)
- Analogy: industrial assembly line

Pipelining

- Multicycle computer execution:



Program Exeuction Order (in instructions)

Time (in clock cycles)

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10
lw $10, $20(1)	Instruction fetch	Instruction decode	Execution	Data access	Write back					
sub $11, $2, $3						Instruction fetch	Instruction decode	Execution	Data access	Write back

- Can apply to our five MIPS stages of fetch-execute

Copyright figure removed

Designing Instruction Sets for Pipelining

These factors help in implementing pipelining:

- All MIPS instructions are same length (simplifying instruction fetch and decode)
- Few instruction formats, source fields in same place (so register operands can be fetched in parallel with instruction decoding)
- Memory operands only in loads and stores (so only one memory access per instruction, and address can be computed in execute stage)

Pipeline Hazards (Overview)

- Hazard: event that blocks normal flow of instructions through pipeline
- Structural hazard example: with single instruction/data memory, instruction fetch cannot overlap with load/store

  Solution: separate instruction and data memories again
- Control hazard example: conditional branch instruction may change sequence of instructions executed
- Data hazard example: result of one instruction is needed by next instruction

Solving Control Hazards (Overview)

- First solution: stall

  Assume enough hardware to decide branch and branch address after instruction decode

Copyright figure removed

- Adds one additional clock cycle to all branches

Better Solutions

- Assume branch failure, stall only on success

Copyright figure removed

- Even better solution: dynamic prediction (based on past history)
- Another approach: delayed decision (always execute next instruction after branch)

Notation

- Use symbols to represent stages of pipeline

Copyright figure removed

- Shading indicates a stage uses the hardware
- Half-shaded means hardware used in first/second half of clock-cycle

Solving Data Hazards (Overview)

Copyright figure removed

- New value of `$s0` not available from register file in time
- Solution: forwarding – take value as soon as it is ready, before it is written to register file

Forwarding With Load-Use Hazard

Copyright figure removed

- Stall is necessary with load-use hazard

Review of Single-Cycle Datapath

Copyright figure removed

Flow is left to right, except for PC update and register writeback

Pipelined Version of Datapath

Copyright figure removed

- Registers hold all necessary intermediate values
- There is a deliberate bug in this design

Corrected Version of Datapath

Copyright figure removed

Pipelined Control

- First, we ignore hazards
- No need for control signals for PC and new pipeline registers
- Each control line is associated with a component active in only one stage
- We need to extend pipeline registers to pass control signals down to where they are used

Copyright figure removed

Data Hazards and Forwarding

Copyright figure removed

Last four instructions use data written by first instruction

NOP solution

- There is a cheap way to get rid hazards without additional hardware – have the compiler put in `nop` instructions.

```
sub $2,$1,$3
and $12,$2,$5
or $13,$6,$2
add $14,$2,$2
sw $15,100($2)
```

- An example of a `nop` is `sll $0, $0, 0`.
- But this increases the running time of the program.

Dependency Detection

- Need notation for pipeline registers
- “ID/EX.RegisterRs” means the name of the register indicated in the rs field of instruction currently in ID/EX
- Example: if destination register of one instruction is first source of the next instruction
- Condition is

  EX/MEM.RegisterRd = ID/Ex.RegisterRs
- Need to also check EX/MEM.RegWrite = 1
- Ignore `$0` as a destination

Forwarding

- First hazard condition:
  if (EX/MEM.RegWrite
  and (EX/MEM.RegisterRd $\neq$ 0)
  and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
- Value can be forwarded from pipeline register
- Value is in EX/MEM.ALUOut
- Need to expand MUX for ALU input to allow this possibility

Pipeline Register Forwarding

Copyright figure removed

Control Signals for Forwarding

MUX control	Source	Explanation
ForwardA = 00	ID/EX	ALUin1 from reg file
ForwardA = 10	EX/MEM	ALUin1 from prev ALUOut
ForwardA = 01	MEM/WB	ALUin1 from writeback
ForwardB = 00	ID/EX	ALUin2 from reg file
ForwardB = 10	EX/MEM	ALUin2 from prev ALUOut
ForwardB = 01	MEM/WB	ALUin2 from writeback

Hardware for Forwarding

Copyright figure removed

Complete Forwarding Conditions

- EX Hazard 1a

  if (EX/MEM.RegWrite

  and (EX/MEM.RegisterRd $\neq$ 0)

  and (EX/MEM.RegisterRd = ID/EX.RegisterRs))

  then ForwardA = 10

- MEM Hazard 2a

  if (MEM/WB.RegWrite

  and (MEM/WB.RegisterRd $\neq$ 0)

  and (EX/MEM.RegisterRd $\neq$ ID/EX.RegisterRs)

  and (MEM/WB.RegisterRd = ID/EX.RegisterRs))

  then ForwardA = 01

- 1b,2b are similar, but with Rt, ForwardB

Datapath to handle forwarding

Copyright figure removed

Data Hazards and Stalls

Copyright figure removed

Load-use hazard requires a stall

The Effect of a Stall

Copyright figure removed

Detecting Hazard Requiring Stall

- Instruction in ID/EX register must be Load; can check MemRead signal
- Either source (rs or rt) of instruction in IF/ID must use same register as Load rt register

```
if (ID/EX.MemRead
and ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
        (ID/EX.RegisterRt = IF/ID.RegisterRt)))
```

Implementing Stall

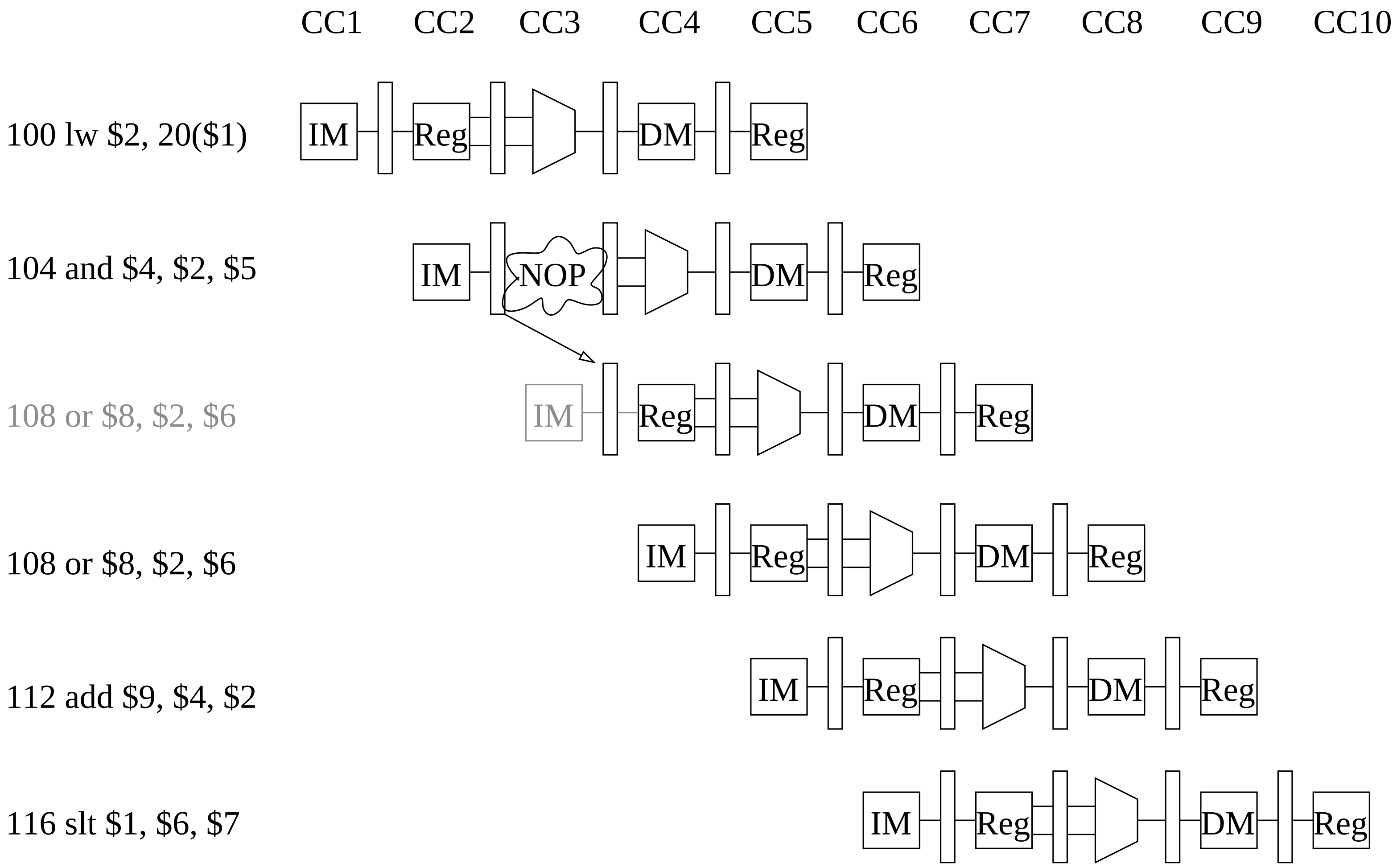
- Must prevent PC and IF/ID registers from changing
- This stalls instructions in IF and ID stages
- Must create bubble in EX stage
- Setting all nine control signals to 0 in EX,MEM,WB control fields of ID/EX register does this

Hardware for Stall

Copyright figure removed

Signed-immediate and branch hardware omitted for clarity

The Real Effect of a Stall



100 lw $2, 20($1)

104 and $4, $2, $5

108 or $8, $2, $6

108 or $8, $2, $6

112 add $9, $4, $2

116 slt $1, $6, $7

Branch Hazards

Copyright figure removed

Branching Earlier in Pipeline

- Branch success decided in MEM stage
- At this point instructions in IF, ID, EX stages must be flushed
- Can move branch execution earlier, to ID stage
- Easy to move branch address calculation to ID stage
- Need to move equality test to ID stage, avoid ALU

Hardware to Execute Branch in ID Stage

Copyright figure removed

Note: this figure is not in the text.

Branch Hazards

Copyright figure removed

Note: this figure is not in the text.

If branch taken, should Instruction 44 be executed?

Branch Flushing

- After moving Branch up to ID stage, instruction after branch will still always execute.

  How to handle this (possibly) errant instruction?

- MIPS forces compiler to handle errant instruction

  Code rearrangement, NOP

- Alternatively, use Branch Flushing

  Zeroing control bits of instruction in IF/ID register will flush IF stage

Hardware for Branch Flushing

Copyright figure removed

Performance of Pipelined Design

- Assume `gcc` mix of instructions

  22% loads, 11% stores, 49% R-format, 16% branches, 2% jumps
- Assume half of all loads followed by use
- Assume quarter of all branches mispredicted
- Average number of cycles per instruction (CPI) is:

  $0.22 \times 1.5 + 0.11 \times 1 + 0.49 \times 1 + 0.16 \times 1.25 + 0.02 \times 2 = 1.17$

Compiler Issues

MIPS architecture with forwarding, stalls guarantees correct execution of MIPS assembly. But...

- Consider the C code:

```
c = 2*a+b;
```

- Straightforward pseudo-MIPS code:

```
lw a
add t,a,a
lw b
add t,t,b
sw c,t
```

- How long does this take to execute?
  Can we do better?

Loop unrolling

- C code:

```
sum = 0;
for (i=0; i<100; i++) {
  sum += a[i];
}
```

- Pseudo-MIPS code (on CPU that stalls on branch taken):

```
    add sum,$0,$0
    add i,$0,$0
    lw t,a[i]
    add sum,sum,t
    addi i,i,1
    bne i,100,-4
    nexti
```

Code Rearrangement Guidelines

- Rearranged code should not affect “behaviour”

  Assembly instruction level

  High level construct level

- Guidelines:
  - Don’t swap lines of code with dependencies:

    ```
    100:  lw $1,100($2)
    104:  add $4,$1,$3
    ```

  - Don’t swap in or out of loops

    ```
        add i,$0,$0
        add sum,$0,$0
        lw t,a[i]
        add sum,sum,t
        addi i,i,1
        bne i,100,-4
        nexti
    ```

Exceptions

- Interrupt: external event unexpectedly changing control flow
  I/O
- Exception: internal event unexpectedly changing control flow
  Arithmetic overflow
- Some books use *interrupt* for both concepts;
  this book used *exception* for both concepts
- OS will process exception
  To handle exception,
  - Save PC for interrupted instruction
  - Remember what caused exception

What caused the exception

- Two general ways to note what caused the exception:
  - *Cause register*: status register that holds field noting cause of exception

    OS checks cause register to process exception
  - *Vectored interrupts*: table of addresses, one for each type of exception

    When interrupt A occurs, branch to routine that processes interrupt A

    OS can determine exception type by looking up address in table
- MIPS uses cause registers

Hardware for Exception Handling

Copyright figure removed

Exception Example

Assume this code executes (addresses in hex)

```
40 sub $11, $2, $4
44 and $12, $2, $5
48 or $13, $2, $6
4C add $1, $2, $1 # This is going to cause the exception
50 slt $15, $6, $7
54 lw $16, 50($7)
```

and exception handler code is (addresses in hex)

```
40000040 sw $25, 1000($0)
40000044 sw $26, 1004($0)
```

Memory Hierarchies

- Readings: Chapter 5, sections 5.1–5.8
- Goal: create illusion of unlimited fast memory
- Problem: faster memories are more expensive, and larger memories can be slower
- Solution: move items to smaller, faster memory automatically when they are needed
- Rationale: locality of reference
  - Temporal: Once accessed, likely to be accessed again soon
  - Spatial: Items “nearby” also likely to be accessed

Copyright figure removed

- Items not in top level are brought up when requested
- Data only copied between adjacent levels (focus on two)
- Block: minimum unit of information present/not present
- Hit: information present when requested
- Miss: information not present, must be copied up
- Terminology: hit ratio, hit time, miss penalty

Caches

- Cache: level of memory hierarchy between CPU and main memory
- Important questions:
  - How do we know if a data item is in the cache?
  - How do we find it?
- Direct mapped: each memory location is mapped to exactly one location in the cache
- Typical mapping: (block address) modulo (number of blocks in cache)
- Example: block = 1 word, cache size = 8, just take lower 3 bits of word address
- Need tag for each block in cache giving original location

Direct mapped cache

Memory Access

Dec	Binary	Hit/miss	Cache block
20	10100		
18	10010		
20	10100		
18	10010		
22	10110		
7	00111		
22	10110		
28	11100		

Cache

Index	V	Tag	Data
000			
001			
010			
011			
100			
101			
110			
111			

Testing For Cache Hit/Miss

Copyright figure removed

Handling Cache Misses

- On miss: stall entire processor until item fetched
- On write: usually written item goes into cache
- Write-through: immediately write item back into memory
- Write-back: write item only into cache

  Cache and memory temporarily inconsistent

  Write back cache block only when it must be replaced
- Could have separate instruction and data caches (increases bandwidth)

Spatial Locality: Larger Block Sizes

Copyright figure removed

Tradeoffs in Choosing Block Size

- Smaller blocks mean more misses for local references
- Larger blocks mean fewer blocks in cache, premature bumping
- Larger blocks increase miss penalty
- Memory must be read on write miss if block size > 1

Block Size Read Example

	Tag	00	01	10	11
000					
001					
010					
011					
100					
101					
110					
111					

	Dec	Tag	Indx	Block	Byte	Hit/Miss
`lw`	48	000	011	00	00	
`lw`	52	000	011	01	00	
`lw`	56	000	011	10	00	
`lw`	60	000	011	11	00	

Block Size Write Example

	Tag	00	01	10	11
000					
001					
010					
011					
100					
101					
110					
111					

	Dec	Tag	Indx	Block	Byte	Hit/Miss
`sw`	184	001	011	10	00	
`lw`	188	001	011	11	00	

Block Size Read/Write Example

	Tag	00	01	10	11
000					
001					
010					
011					
100					
101					
110					
111					

	Dec	Tag	Indx	Block	Byte	Hit/Miss
`lw`	48	000	011	00	00	
`lw`	52	000	011	01	00	
`lw`	48	000	011	00	00	
`sw`	184	001	011	10	00	
`lw`	188	001	011	11	00	
`lw`	48	000	011	00	00	

Block Size Code Example: Arrays

Pseudo-MIPS: Sum the elements in an array A

```
      100 add $1, $0, $0
      104 add $2, $0, $0
loop: 108 lw $3, A($1)
      112 add $2, $2, $3
      116 addi $1, $1, 1
      120 addi $4, $1, -1000
      124 bne $4,$0 loop
      128 nop
```

Assume 1 cc per instruction (data forwarding, etc)

lw takes 1cc if in cache

Block size 1: fetch from memory takes 17cc extra

Block size 4: fetch from memory takes 20cc (4 words) extra

Block Size Code Example: Linked List

Pseudo-MIPS code: sum elements in a linked list

```
      100 lw $1, A
      104 add $2, $0,$0
      108 beq $1,$0, done
loop: 112 lw $3, 0($1)
      116 add $2, $2,$3
      120 lw $1, 4($1)
      124 bne $1,$0, loop
done: 128 nop
```

Assume 1 cc per instruction (data forwarding, etc)

lw takes 1cc if in cache

Block size 1: fetch from memory takes 17cc extra

Block size 4: fetch from memory takes 20cc (4 words) extra

Alternate Placement Schemes

- Fully-associative: a block can go anywhere in the cache
- Requires a comparator for each cache entry
- Set-associative: combines two ideas

Copyright figure removed

A 4-way Set-Associative Cache

Copyright figure removed

2-way set associative cache

Memory Access

Dec	Binary	Hit/miss
20	10100	
18	10010	
20	10100	
18	10010	
22	10110	
7	00111	
22	10110	
28	11100	

Cache

Index	Tag0	Tag1
00		
01		
10		
11		

4-way set associative cache

Memory Access

Dec	Binary	Hit/miss
20	10100	
18	10010	
20	10100	
18	10010	
22	10110	
7	00111	
22	10110	
28	11100	

Cache

Index	Tag0	Tag1	Tag2	Tag3
0				
1				

Fully associative cache

Memory Access

Dec	Binary	Hit/miss
20	10100	
18	10010	
20	10100	
18	10010	
22	10110	
7	00111	
22	10110	
28	11100	

Cache

Tag0	Tag1	Tag2	Tag3	Tag4	Tag5	Tag6	Tag7

Mixing "Ways" and "Blocks"

- Can have both ways and blocks in a cache

  Example: 2-way set associative, 4 word line size

	Tag0	Block 00	Block 01	Block 10	Block 11	Tag1	Block 00	Block 01	Block 10	Block 11
0000										
0001										
0010										
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

- Caches typically associative with block size > 1

How useful are the ways?

- For small (8 word) caches, fully associate makes sense
- For larger caches, more “ways” means fewer cache misses, but is the extra hardware worth it?
- Experiment on a 64KB cache, 16-word block:

Associativity	Miss Rate
1	10.3%
2	8.6%
4	8.3%
8	8.1%

Data cache miss rates

Copyright figure removed

Calculating Average Memory Access Time

- How long to access memory?

  AMAT = Time for a hit + Miss rate × Miss penalty

- Example

  1ns clock, miss penalty 20cc, miss rate 0.05 per instruction, cache access time of 1cc

Sorting Example

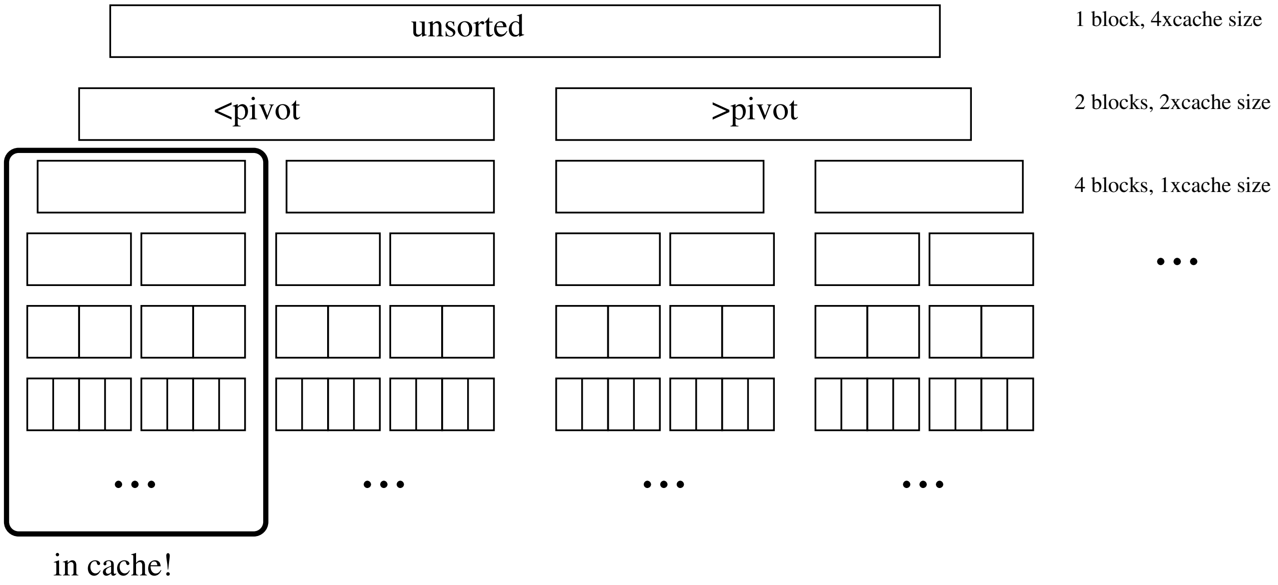
Impact of cache on performance

Copyright figure removed

- Left: instructions
- Center: run time
- Right: cache misses

Thus, although radix sort takes fewer instruction, quicksort is faster because of fewer cache misses

Quicksort



- Initially, data too large for cache...
- ...but at some depth, data fits in cache and remains in cache for all deeper levels

Virtual Memory

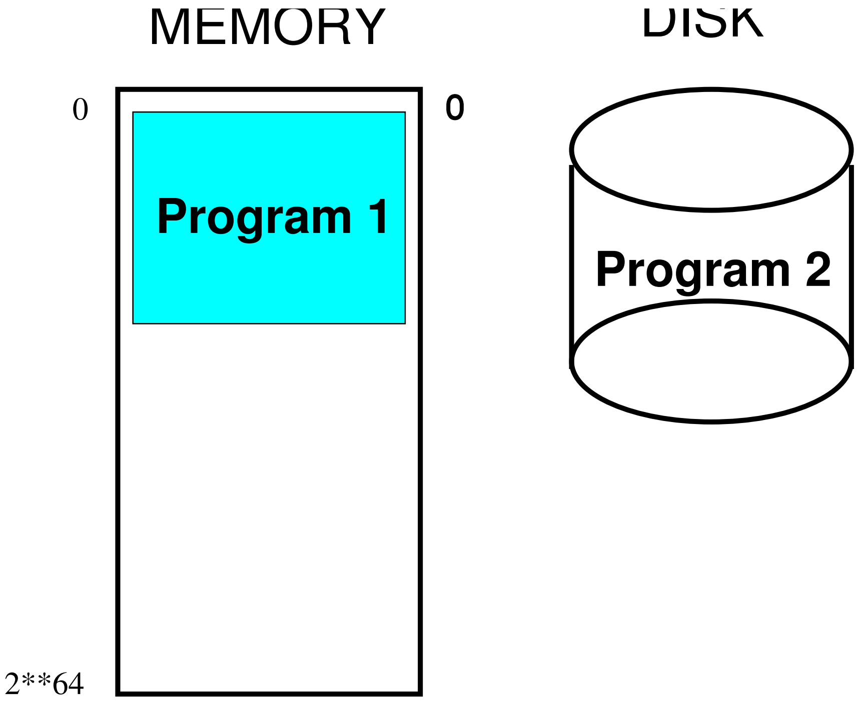
- Level of memory hierarchy between main memory and secondary storage (disks)
- Motivations:
  - Sharing memory among multiple programs
  - Allowing single user program to exceed size of main memory
- Different terminology:
  - Page: virtual memory block
  - Page fault: virtual memory miss
- Idea similar to cache:
  - Complete program (instructions and data) stored on disk
  - Only keep parts you need in memory
  - Use large (4KB+) blocks to reduce costs

How to run two programs at once?

- Each program starts at address 0 and goes up to $2^{64} - 4$ (64 bit)
- How do we run two programs?
- Time slice: each program gets to run for a while, then another program runs.
- How do we switch between programs?

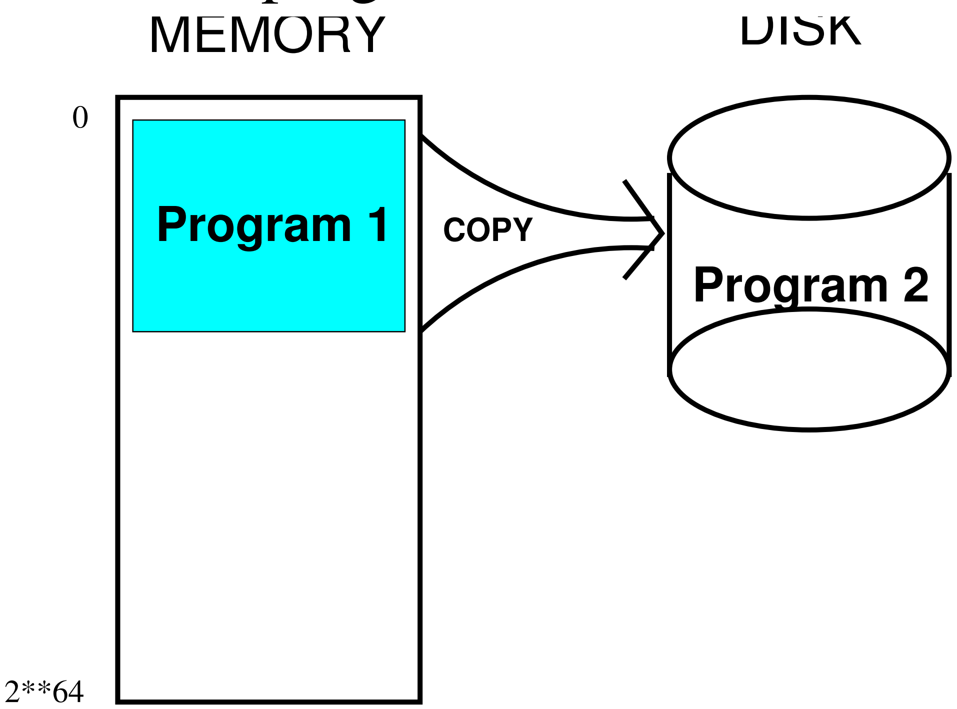
Idea 1: Copy back/forth to disk

- When changing programs, copy current program onto disk and read next program from disk



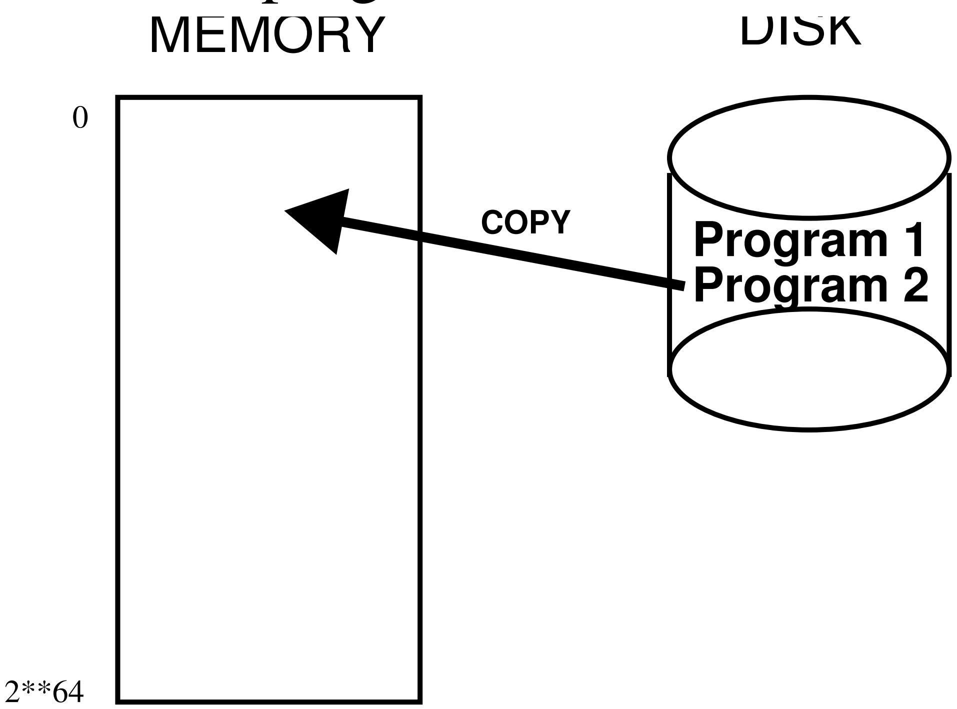
Idea 1: Copy back/forth to disk

- When changing programs, copy current program onto disk and read next program from disk



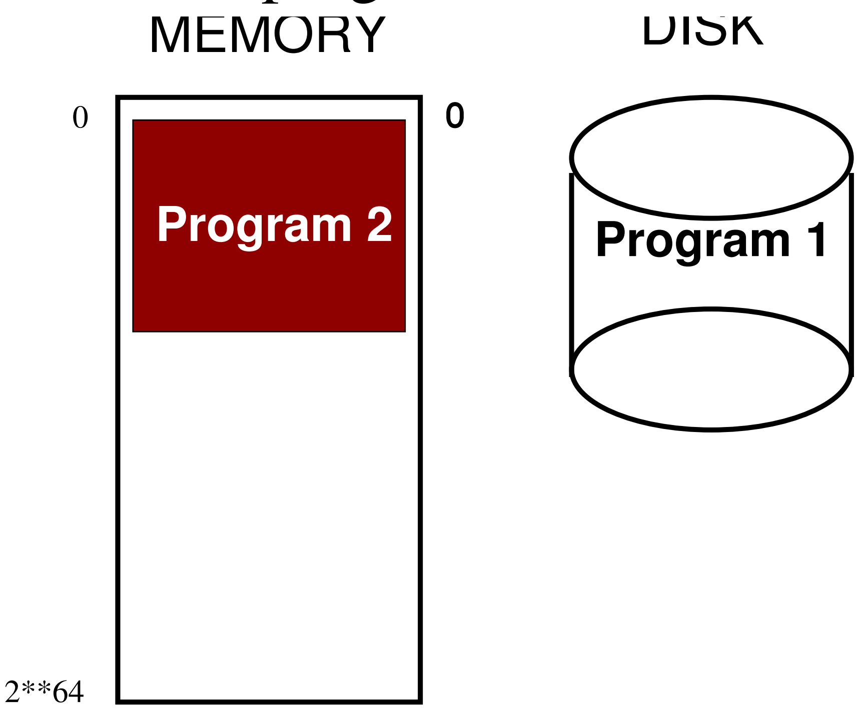
Idea 1: Copy back/forth to disk

- When changing programs, copy current program onto disk and read next program from disk



Idea 1: Copy back/forth to disk

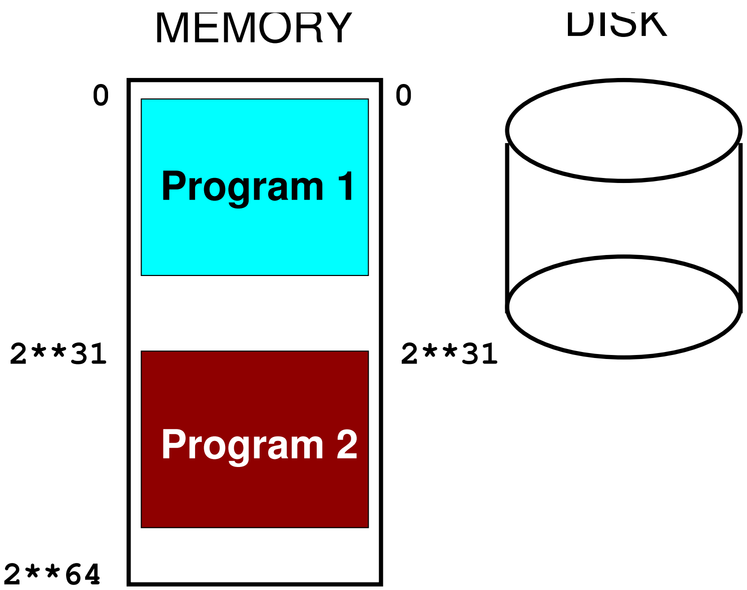
- When changing programs, copy current program onto disk and read next program from disk



- At 70MB/s, swapping between two 100MB programs takes over 2 seconds

Idea 2: Keep both programs in memory

- Keep both programs in memory as follows:
  - Assume 4GB+ memory
  - Assume 2GB program size
  - Program 1: 0 to $2^{31} - 4$
  - Program 2: 2^{31} to $2^{32} - 4$

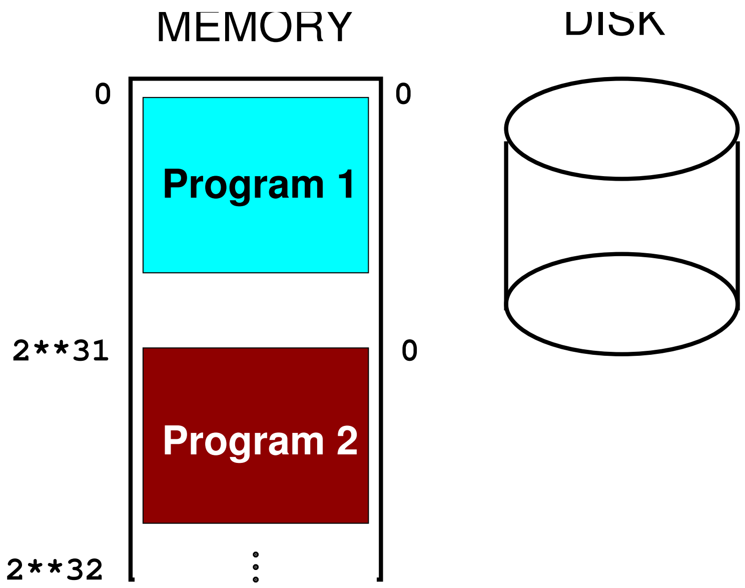


- Compiler creates Program 2’s address space to start at 2^{31}
- When swapping, just start second program.
- Problem 1: What if two programs overlap.
- Problem 2: What if both programs need more than 2GB?

Idea 3: Keep both program in memory

- Keep both programs in memory as follows:
  - Assume 4GB+ memory
  - Assume 2GB program size
  - Program 1: 0 to $2^{31} - 4$
  - Program 2: stored in 2^{31} to $2^{32} - 4$

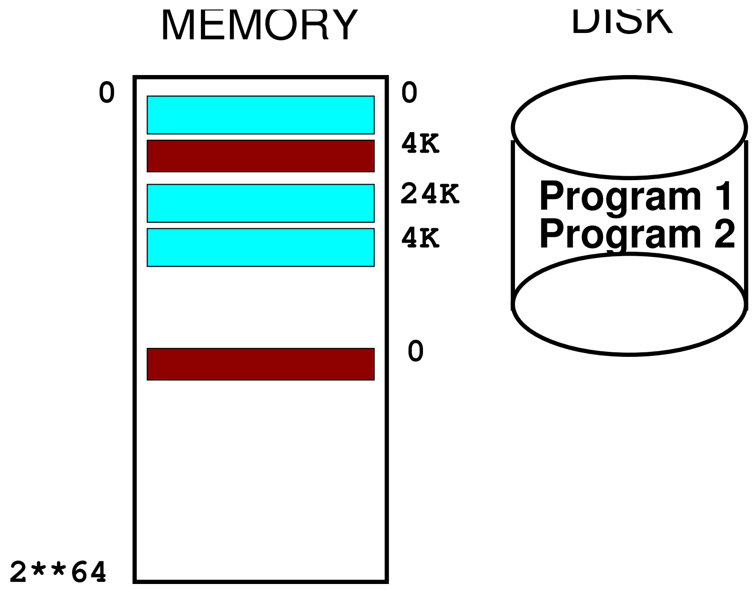
    BUT program addresses from 0 to $2^{31} - 4$



- When swapping, just start second program.
- Problem 1: Program 2's addresses need to be converted to physical addresses
- Problem 2: What if both programs need more than 2GB?
- Problem 3: Slow to load large programs

Idea 4: Virtual Memory

- Like idea 3, but split memory into pages (eg, 4KB pieces).
- Each page of program can be in any page of physical memory.



- Use page table to map program address to physical address.
- If program wants more space, give it a new page.

  If no pages available, replace ”lightly used” page

Virtual Memory Mappings

Copyright figure removed

Copyright figure removed

Key Decisions for Virtual Memory

- Misses very expensive (millions of cycles)
- Pages are larger (4KB up to 64KB)
- Fully-associative schemes pay off, though full search too expensive
- Page faults handled in software (OS)
- Write-back used (write-through too expensive)

Page Replacement Schemes

When a page needs replacement, we must decide which to replace.

- LRU (Least Recently Used)
- LFU (Least Frequently Used)
- FIFO (First In, First Out)
- Random

Page Tables

Copyright figure removed

Page Table Example

4KB pages

Page Table

	V	Physical Address
0000 ... 0000 0000 0000 0000	1	00 0000 0000 0000 0011
0000 ... 0000 0000 0000 0001	1	01 0000 0001 0000 0101
0000 ... 0000 0000 0000 0010	0	11 0000 1010 0000 0000
0000 ... 0000 0000 0000 0011	1	00 0000 0000 0000 0001
...		...

Virtual Address	Physical Address
0000 ... 0000 0000 0000 0000 0100 1000 1111	
0000 ... 0000 0000 0000 0001 0000 0000 0001	
0000 ... 0000 0000 0000 0010 0111 1000 1010	

More on Page Tables

- Page replacement policies:
  - Optimal: replace page used farthest into the future
  - Good approximation: LRU (Least Recently Used)
  - Too expensive to implement exactly, use approximation (eg reference bits)
- Page tables are large (contain mappings for every virtual page)
  - Multi-level page tables reduce space needed
  - Page tables can themselves be paged
- VM must use writeback (called copyback here)
  - Dirty bit

A Problem With Virtual Memory

- Convert virtual address to physical address:
  look up virtual address in Page Table
- Page Table stored in memory
- To do `lw A`, we have to
  - Look up virtual address `A` in Page Table to get physical address `A′`
  - Read `A′` from physical memory

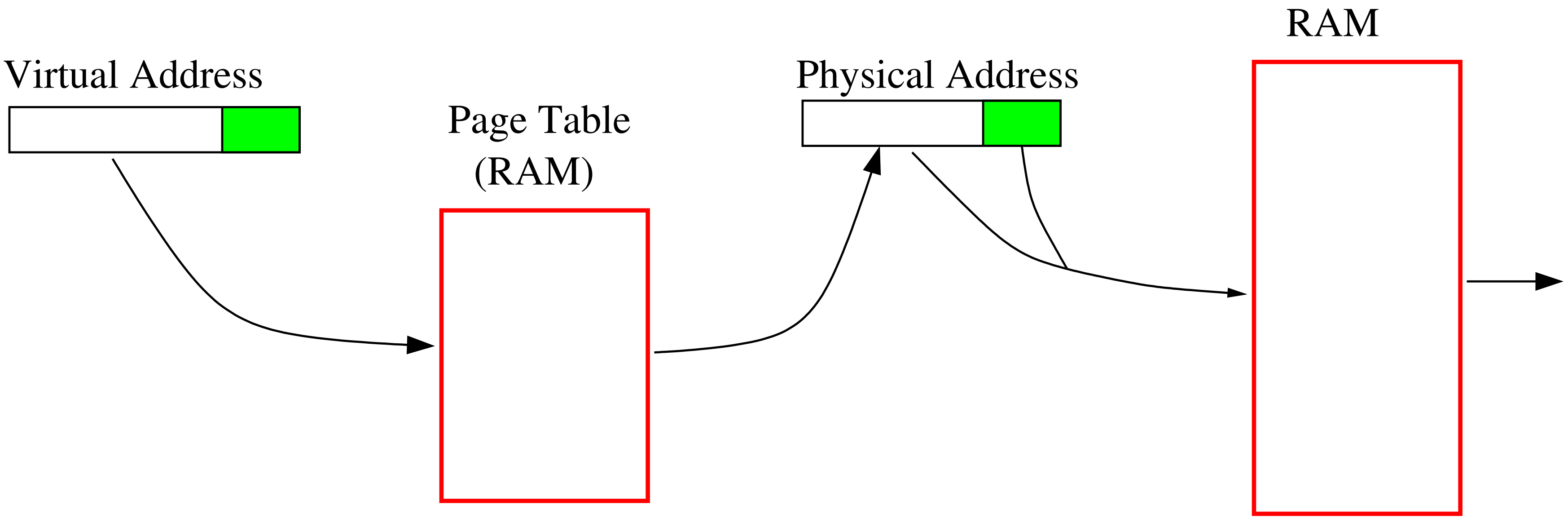
  With virtual memory, `lw` requires **two** memory accesses

Translation Lookaside Buffers

Copyright figure removed

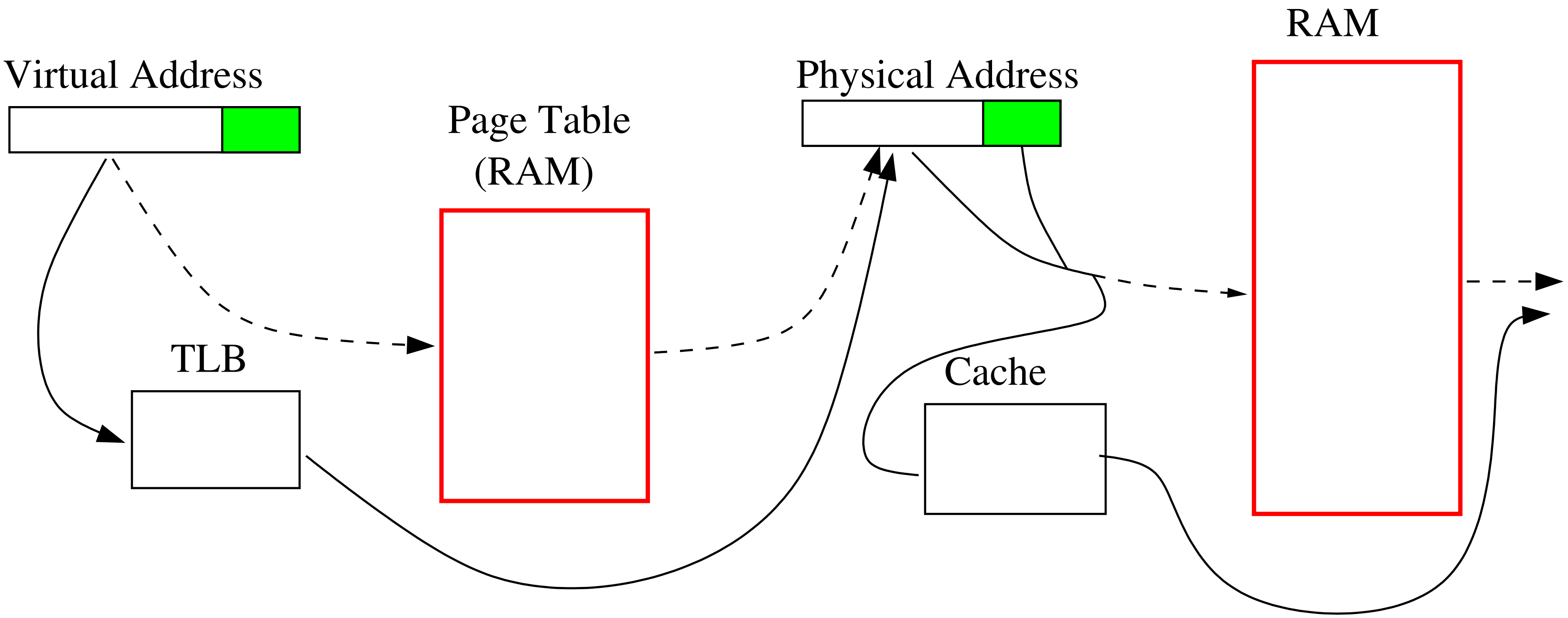
A cache for page table entries

Virtual Memory: Two Memory Accesses



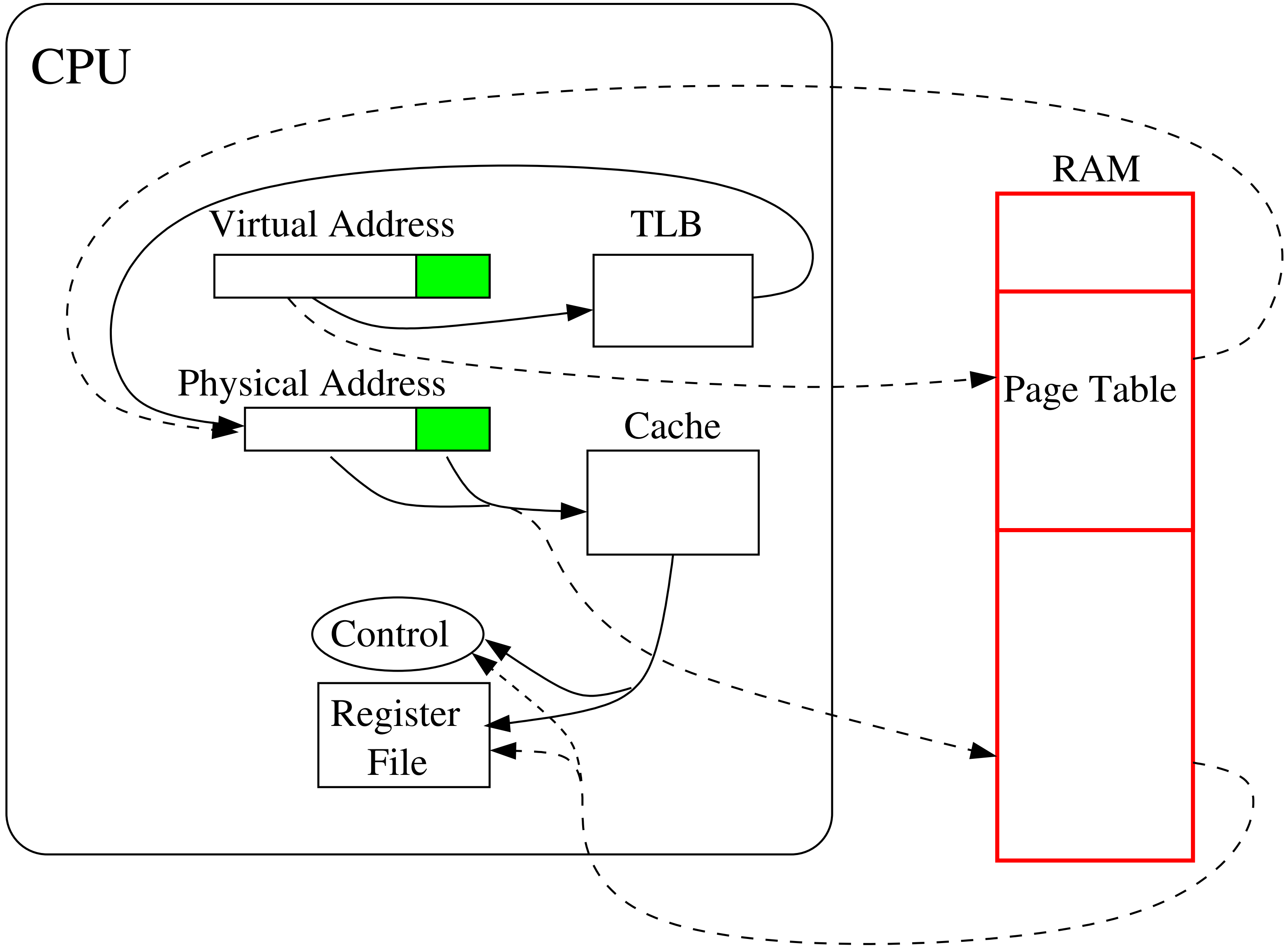
Reading memory from a virtual address requires two memory accesses

Virtual Memory: TLB and Cache



Reading from the TLB and cache avoids reading from memory

Virtual Memory: The CPU



VM and Multitasking

- Need two modes: user process or operating system process
- OS processes are privileged (can do more)
  - Write page table register
  - Modify page tables or TLB
- Need method of switching modes
  - User process to OS: System call or exception
  - OS to user process: return-from-exception
- To swap processes, OS rewrites page table register and flushes TLB

Input/Output

- Readings: Chapter 1.4, 4.9, 5.2, 5.5, 5.11.
- Our interests:
  - Concepts rather than numerical details
  - Demands on CPU
  - Connection of peripherals to CPU

Typical Organization of I/O Devices

Copyright figure removed

Demands of I/O Systems

- Supercomputers:
  - Single large read at start of batch job
  - “Checkpointing” saves during run of job
- Transaction processing:
  - Many small changes to large database
  - Need good throughput but also reliability
- Unix file systems:
  - Mostly to small files
  - More reading than writing
- Benchmarks have been developed for all three areas

Types and Characteristics of I/O Devices

- Characteristics:
  - Behaviour: input, output, storage
  - Partner: human or machine
  - Data rate
- Some types:
  - Keyboard: input device, human partner, data rate 10 bits/sec (15 WPM)
  - Graphics display: output device, human partner, data rate 60 Mb/sec
  - Disk: storage device, machine partner, data rate 100Kb/sec—200MB/sec

Keyboards and Mice

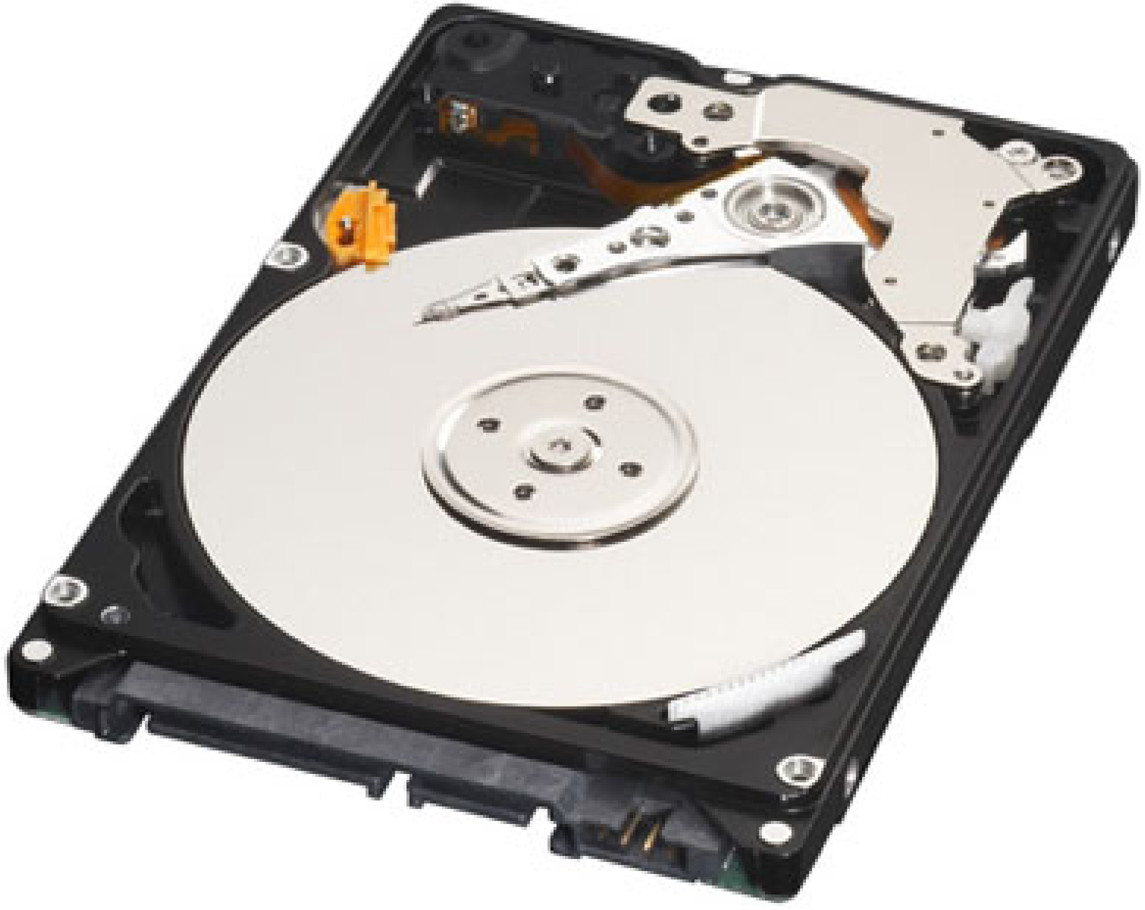
- Keyboards slow enough that each keystroke can generate interrupt serviced by CPU
- Alternate arrangement: bytes generated by keystrokes can be put into small buffer that is checked periodically by CPU
- We say the CPU polls the device
- A mouse needs to be polled more often to ensure smooth animation of cursor on screen
- Mouse generates pulses indicating small movements in four directions (up,down,left,right)

Magnetic Disks

Copyright figure removed

Typically 1-4 platters, 10,000-50,000 tracks, 100-500 sectors, Pre 2011: 512 bytes per sector. Since 2011: 4KB per sector.

Inside a Magnetic Disk

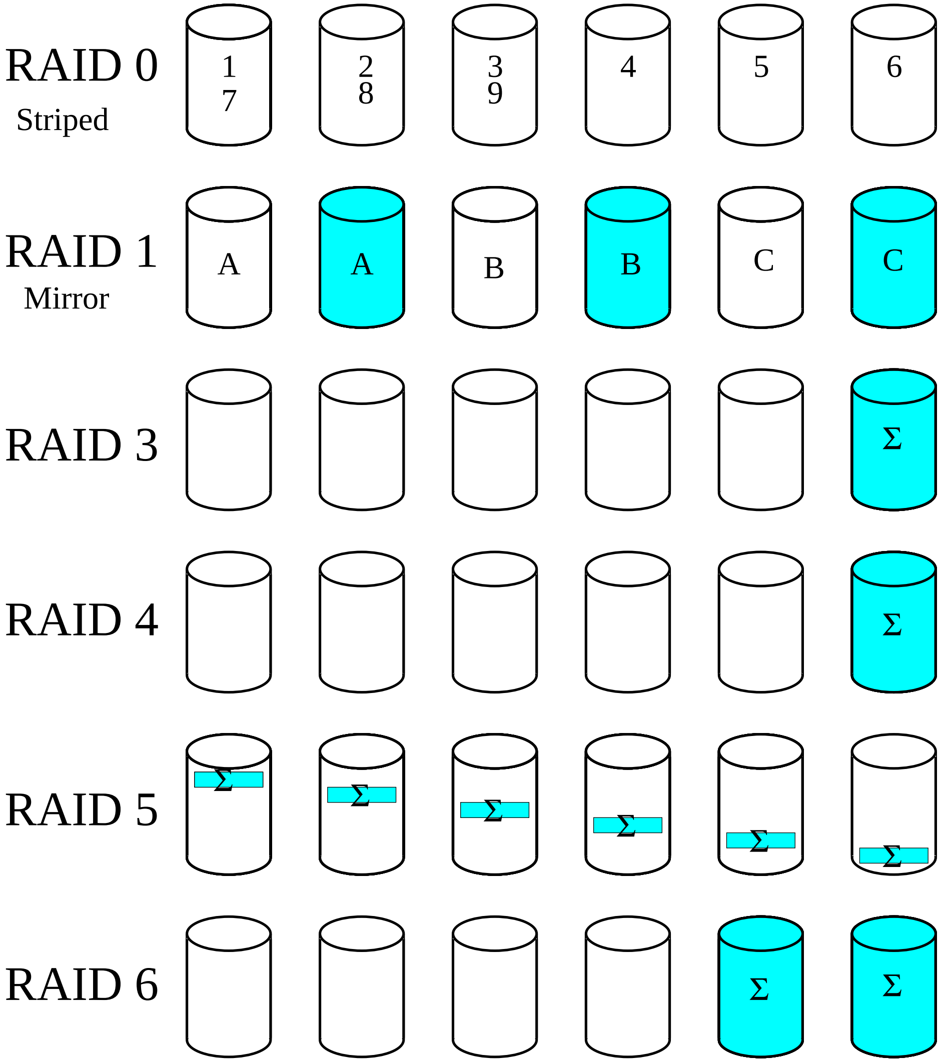


Characteristics of Magnetic Disks

- Seek time: time to position read/write heads over desired track
- Rotational latency: must wait for correct sector to rotate under head
- In practice these two depend highly on mix of requests
- Transfer time: time to read block of bits (typically sector)
- Can add cache to lower these times
- RAID: Redundant Array of Inexpensive Disks

  Use redundancy and coding to improve throughput and reliability

RAID—Redundant Arrays of Inexpensive Disks


RAID 0
Striped

RAID 1
Mirror

RAID 3

RAID 4

RAID 5

RAID 6

Blue disks marks redundant information.

Flash Memory/SSD

- HDD may be reaching limits
  15,000 RPM available; 20,000 RPM has problems
- Flash memory similar to DRAM:
  put voltage on line and test, but doesn't forget
- Two types of cells:
  SLC flash: 2 values, 1 bit/cell
  MLC 4 values, 2 bits/cell
- Comparison:
  SLC faster (25μs vs 50 μs)
  MLC stores twice as much

Solid State Drive (SSD)

- SSD is groups of flash cells

  Cells are grouped into pages (4KB)

  Pages grouped into blocks (512KB)
- Read and write into page
- Erase a block (128 pages in a block)
- Speed comes from parallelism

  Flash memory: 20MB/s

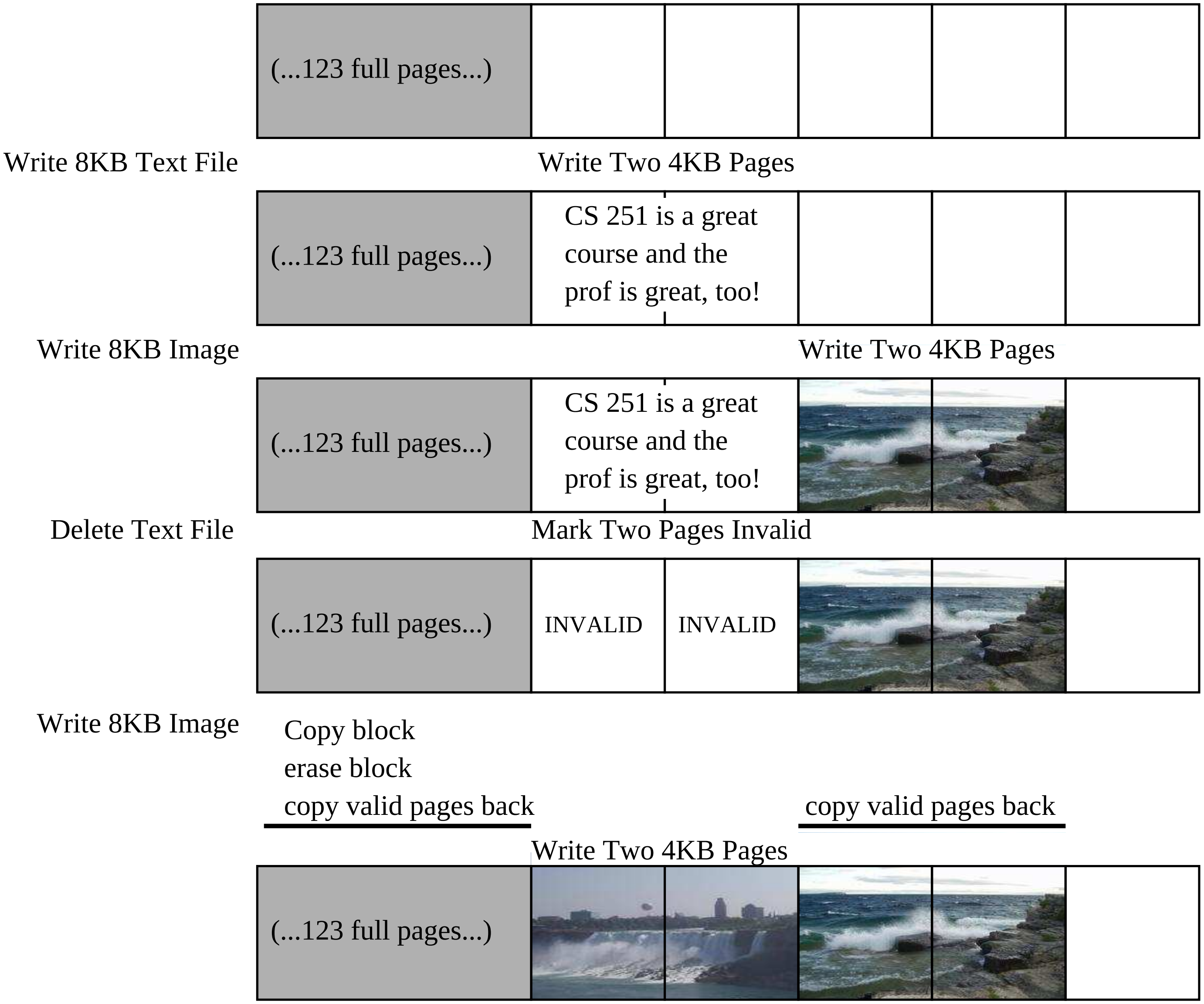
  SSD: 10 in parallel: 200MB/s

SSD Issues

- Characteristics:
  - Can read and write pages
  - Can ONLY erase BLOCKS
  - Can NOT overwrite page:
    have to erase first
- Result: Controller writes to every page before erasing
  Once "full", a 4KB page write requires
  - Read 512KB block into temp
  - Erase 512KB block
  - Write back valid pages
- Result: SSD fast when new, but later is slow (to write)
  OS writes lots of temp files, so OS appears to slow down
  (OS has properly handled this since Windows 7.)

SSD Write Example

Starting with 123 pages full and 5 empty...



(...123 full pages...)

Write 8KB Text File

Write Two 4KB Pages

(...123 full pages...)

CS 251 is a great
course and the
prof is great, too!

Write 8KB Image

Write Two 4KB Pages

(...123 full pages...)

CS 251 is a great
course and the
prof is great, too!

Delete Text File

Mark Two Pages Invalid

(...123 full pages...)

INVALID

INVALID

Write 8KB Image

Copy block
erase block
copy valid pages back

copy valid pages back

Write Two 4KB Pages

(...123 full pages...)

Comparison (Fall 2011 vs Fall 2013)

	2011			2013			Announced
	Size	Price	$/TB	Size	Price	$/TB	Size
HHD	3TB	$180	$60	4TB	$200	$50	6TB
SSD	250GB	$400	$1600	1TB	$570	$570	1.6TB

- Observations:
  - SSDs increasing in capacity quicker than HHDs
  - SSDs price per unit storage dropping faster than HHDs
- Speed:

  SSD range from 100 times slower to 10 times faster

  (newer SSD's may not be slower than HHD any more)
- Hybrid models

  HDD with 100GB SSD used as cache
- SSDs are appearing in large data centers (Amazon, Facebook, Dropbox)

Networks

- Ethernet: one-wire bus, 10/100/1000Mbit/sec transfer rate, no central control
- Nodes listen for silence, then transmit
- Collisions resolved by random backoff
- Basis for most LANs (local area networks) when combined with some switching
- Long-haul networks: distances of 10 to 10,000 km
- Usually packet-switched: information broken up, reassembled
- Protocols for reliable delivery (TCP) and addressing (IP)

Buses

Copyright figure removed

A typical output operation (defined from processor point of view)

Bus Parameters

- Buses are typical bottleneck for I/O throughput
- Limitations on speed: length, number of devices
- Types: processor-memory, I/O, backplane
- Communication schemes: synchronous, asynchronous
  - Synchronous have clock connected to control lines, fixed protocols

    Fast, but require similar devices and must be short
  - Asynchronous buses require “handshaking” communication protocol

    Handled as two communicating FSMs

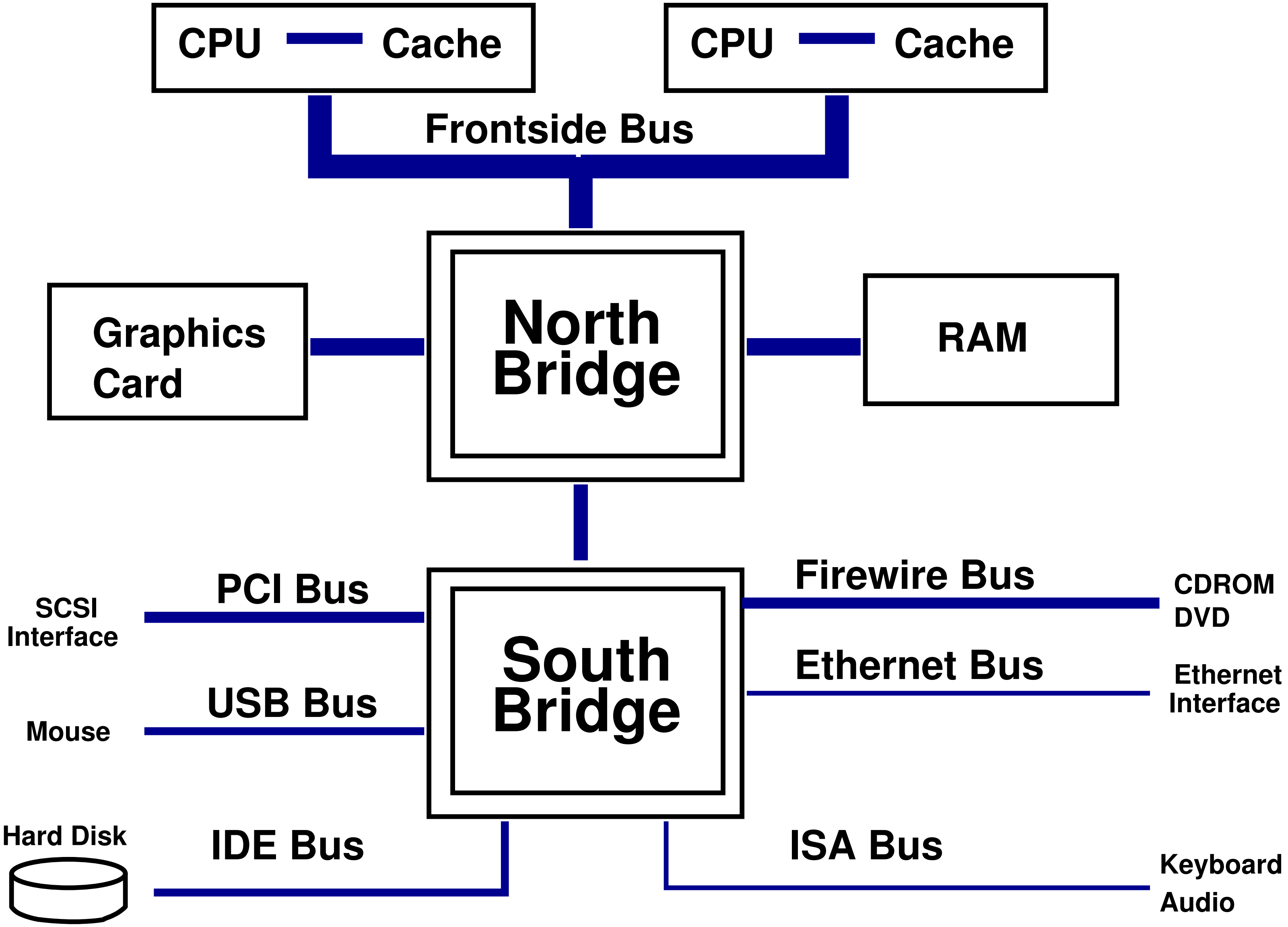
Bus Arbitration

- Who decides who gets access to a shared bus?
- Simplest scheme: the processor
- Better: have multiple bus masters
- Choosing among bus masters is bus arbitration
- Need fairness but also scheme of priorities
- Typical schemes: daisy-chain, centralized, token-passing, self-selection, collision detection (as in Ethernet)

Interfacing with Processor and Memory

- Two schemes for processor I/O: memory-mapped, special instructions
- Polling versus interrupt-driven I/O
- Usually need to have different levels of interrupts depending on priority
- DMA: direct memory access (device controller transfers data right into memory without processor involvement)
- Creates problems with caches or VM

Example: Pentium Bus



Multiprocessors

- Readings: Chapter 6
- Motivations:
  - Increased throughput
  - Scalability
  - Fault tolerance
- Issues:
  - How to share data among processors?
  - How to coordinate processing?
  - What is the optimal number of processors?
- Terminology:

  Rather than ”multiprocessor”, the term *multicore* is used

Parallelism is Easy!

- Modern uniprocessor (single core) CPUs are parallel computers

  Pipelining
- Pipelining gives important but limited benefits
- Want to do better:

  break task into multiple pieces,

  run on multiple cores

Parallelism is Hard!

- How to program multicore machines?
- Large gains needed to justify expense

  Book example: to get a factor of 90 speedup with 100 cores, program must be “99.9%” parallelizable
- To achieve large gains, must balance load

  If one processor does more work, rest are idle
- Programming language constructs

  Synchronization, data sharing, data locking, ...

Hardware Multithreading

- Thread: process of execution
  - Potentially, one program can have multiple threads of execution (parallelism)
  - On single core, “swap” between execution of threads
  - On multi-core, one process per core (“swap” if needed).
- Harware multithreading
  - Multiple threads run on a single core
  - Idea is that when one thread stalls, run another thread
- Fine-grained multithreading: switch threads each instruction
- Coarse-grained multithreading: switch threads on long stalls
- Bottom line: hardware multithreading reduces idle time of processor

Some Parallelism is Easy

- Instruction/Data classifications:
  - SISD—Single Instruction, Single Data
    Pentium 4
  - SIMD—Single Instruction, Multiple Data
    MMX, SSE, SSE2
  - (MISD—Multiple Instruction, Single Data)
  - MIMD—Multiple Instruction, Multiple Data
    “algorithmic” parallelism

SIMD

Single **I**nstruction, **M**ultiple **D**ata

- Example:

  ```
  vec_res.x = v1.x + v2.x; vec_res.y = v1.y + v2.y;
  vec_res.z = v1.z + v2.z; vec_res.w = v1.w + v2.w;
  ```

  vs

  ```
  movaps xmm0, [v1]
  addps  xmm0, [v2]
  movaps [vec_res], xmm0
  ```

- Works best on *data-level* parallelism

  Data independent from one another

- Works best with arrays in `for` loops

  Numerical computations (matrix multiply)

- MMX, SSE, AVX, ...

GPU: an extreme example of SIMD

- GPU is high-performance parallel computer
- Intended for graphics, so no need to support general computing
- Relies on hardware multithreading rather than caches to handle memory latency

  Execute other threads while waiting for memory
- Relied on memory bandwidth rather than latency
- Intended for graphics, but support for non-graphics (CUDA)

Example: y=ax+y

Computing y = ax + y with a serial loop:

```
void saxpy_serial(int n, float alpha, float *x, float *y)
{
  for(int i = 0; i<n; ++i)
    y[i] = alpha*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

Computing y = ax + y in parallel using CUDA:

```
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if( i<n ) y[i] = alpha*x[i] + y[i];
}
// Invoke parallel SAXPY kernel (256 threads per block)
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Multiprocessor Parallelism

What’s required to run a program on multiple cores?

- Compiler support

  Compiler recognizes code that can run in parallel (SIMD)

- Programming language support:

  Threads, locks, communication

- Hardware support:

  Shared memory access

  Caches

  Networks, Message passing

Communication: Shared-Memory vs. Message Passing

- Shared-memory view: processors have a single common address space
  - Uniform Memory Access: any reference by any processor takes same time
  - Non-Uniform Memory Access: each processor has “nearby” and “farther away” memory
  - UMA easier to program, NUMA more realistic and scalable
  - Both require methods of synchronization
- Message-passing view: processors have private memories, communicate via explicit messages
  - Synchronization is more natural
  - More suited to network view

Message Passing Multiprocessors

- Some large-scale computers with high-performance message passing networks

  Fast, but expensive

- Some concurrent applications have task level parallelism that require little communication
  - Web search, mail servers, file servers
  - Clusters: an example of message-passing parallel computers
  - Separate OS on each computer, separate memory
  - Separate memories, etc., leads to increased system dependability, expandability

- *Warehouse-scale computing*
  - Internet services may require 50,000–100,000 computers

    Require special building, power, cooling
  - Economy of scale ⇒ cloud computing

Organization: Bus-Connected vs Network-Connected

Copyright figure removed

Copyright figure removed

- Big issue: maintaining data consistency (called cache coherency/consistency)
- Data read by several processors will show up in several caches
- What happens when that data is changed?

Multiprocessor Cache Coherency: Snooping

- Idea: information to maintain coherency is available on bus
- Each processor's cache monitors bus
- Simplest idea: write-update (or write-broadcast)
  - When a processor writes a block, all other caches containing that block are updated with new value
  - Increases bus traffic, limits scalability, but reduces latency
- Better idea: write-invalidate
  - When a processor writes a block, it first issues an invalidation signal
  - Other caches invalidate their copies of that block
  - Subsequent writes by same processor result in no bus traffic
  - Subsequent reads by other processors are slower
- Variations: build in exclusivity (avoiding need for invalidation broadcast), transferring data between caches directly

Network Topologies

Copyright figure removed

Variations: cylinder, 3D torus, mesh of trees; cube-connected cycles

More Network Topologies

Copyright figure removed

Variations: butterfly, shuffle-exchange, Beneš network

Coherency in Networked Multiprocessors

Copyright figure removed

Cache-level coherency

Memory-level coherency

The Evolution-Revolution Spectrum

Copyright figure removed

- Items further right require more effort from programmers/users
- Software, not hardware, is current bottleneck

Overview

Many other ideas in architecture.

We will look at a few:

- Microprogramming

  VAX

  CISC vs RISC

- SPARC Ideas

  Register window

  32-bit jump command

Microprogramming

- Idea: control unit for datapath runs program for each assembly instruction
- Multicycle datapath so far too simple for microprogramming
  - Microprogram needs scratch registers
  - Microprogram needs conditional branches
- Need to extend datapath, modify to give us extra power
  - Extra registers in register file
  - Control lines to specify read/write registers
  - Control specify constants
  - Status bits sent to control

Microprogramming: Control of Datapath

Copyright figure removed

- Note: Signals for the multi-cycle datapath. Microprogrammed computer would need more signals.

Microinstructions

Idea: write a sequence of microinstructions for each assembly instruction

- One microinstruction = one set of signals to datapath
  (One clock cycle)
- Each microinstruction specifies sequencing
- Fields of microinstruction have specified meanings
- Represent values symbolically to make them readable
- Usually very architecture-specific

Microinstructions stored in a ROM
(ROM is effectively a type of cache)

Microinstructions

- Microinstruction similar to assembly language instruction, but usually closer to control lines of hardware
- Each microinstruction specifies how to use the datapath and specifies which is the next microinstruction.
- Control of datapath:
  Similar to other architectures, but
  - Special temporary registers not available to assembly program.
  - Constants embedded in microinstructions
- Next microinstruction choices:
  - FETCH: fetch a new assembly instruction
  - NEXT: execute next microinstruction in sequence
  - JUMP N: goto microinstruction N

Microinstructions for five MIPS instruction

- Microcode for instruction fetch

  Fetch instruction from memory

  Do *n*-way branch to code for actual instruction

- Microcode for R-format, lw, sw, beq, jump

  A different encoding of FSM for multicycle computer

  Assembly instruction example: Add rA,rB,rC:

```
Add1: uAdd rA,rB,rC   FETCH
```

Example: `swap rA, rB`

Exchange the contents of two registers

```
Swap1: uAdd rA,0,rT   NEXT
Swap2: uAdd rB,0,rA   NEXT
Swap3: uAdd rT,0,rB   FETCH
```

Branch Example

- Consider an R-format command like

```
summ $s1=rd,$s2=rs,$s3=rt
```

that sets $s1 to the sum of $s3 words starting at M[$s2].

```
Summ0: uAdd 0,0, rT1          NEXT
Summ1: uBez rT1,0, Summ10     ---
Summ2: uAdd rt,0, rT2         NEXT
Summ3: uAdd rs,0, rT3         NEXT
Summ4: uLw rT3, rT1           NEXT
------------------
Summ5: uSub rT2,1, rT2        NEXT
Summ6: uBeq rT2,0, Summ10     ---
Summ7: uAdd rT3,1, rT3        NEXT
Summ8: uLw rT3, rT4           NEXT
Summ9: uAdd rT1,rT4, rT1      JUMP Summ5
Summ10: uAdd rT1,0, rd        NEXT
Summ11: uJump Start           FETCH
```

rT1: result (rd); rT2: cntrl (rt); rT3: mem addr (rs); rT4: temp

VAX: A microprogrammed architecture

- VAX is an example of a microprogrammed architecture
  Some instructions ran faster than others
- Goals
  - Direct support for high-level language constructs (procedure calls, multi-way branching, loop control, array subscript calculation)
  - Minimize code space to save memory
- Variable length instruction encoding (from 1 to 54 bytes)
- Sixteen 32-bit registers, R0 through R15
- Some are special (eg R15 is PC, R14 is stack pointer)
- Large number of instructions (304) and data types

VAX instruction encoding

- First byte is opcode (or first two)
- Opcode implies number of operands, data type
- For each operand, one byte with addressing mode and register name (4 bits each)
- More information may follow if needed (e.g. constants)
- Example: `ADDL3 R1, 737(R2), (R3)[R4]`
  7 bytes of storage

Examples of complex VAX instructions

- MOVC3: copy a string
- SCANC: scan for character
- AOBLEQ: add one and branch on less than or equal
- INSQUE: insert into queue
- CRC: calculate cyclic redundancy check
- POLY: evaluate polynomial

VAX/MIPS instruction set comparison

- Task: copy fifty words of data from one array to another
- Pseudo-code:

  ```
  for i=0 to 49 do b[i] = a[i];
  ```

- Can write programs in each type of assembler, but comparison requires some assumptions
- Suppose base address of array a is in R3/s3, base address of array b is in R4/s4

MIPS code

```
      addi $t2, $0, 50    # put 50 into t2
      addi $t3, $s3, 0    # copy s3 into t3
      addi $t4, $s4, 0    # copy s4 into t4
loop: lw $t1, 0($t3)      # get word pointed to by t3
      sw $t1, 0($t4)      # put word where pointed by t4
      addi $t3, $t3, 4    # increment t3
      addi $t4, $t4, 4    # increment t4
      addi $t2, $t2, -1   # decrement count
      bne $t2, $0, loop   # loop if count not zero
```

VAX code

```
      CLRL R2                ; R2 is index
LOOP: MOVL (R3)[R2], (R4)[R2]; (note scaling)
      AOBLSS #50, R2, LOOP   ; add one, branch on <50
```

Analysis of MIPS code

- Use pipelined implementation, with all forwarding possible for data hazards, including load-use
- Branch hazards require stall on success
- Code rearrangement to get rid of load/store stall and branch stall
- One change: memory can be slow, whole machine stalls until memory access completes
- Let C be cycle time, M be memory access stall time

Analysis of VAX code

- Assume multicycle implementation similar to microcoded MIPS implementation done in class
- No pipelining
- CLRL, AOBLSS take 4 cycles; AOBLSS is 2-word instruction
- MOVL takes 6 cycles because of operand address calculations
- Above assumptions are reasonable but do not correspond exactly to reality
  - In reality, VAX instructions are fetched by the byte, and the branch would be an 8-bit displacement
  - VAX implementations pipeline the microcode, so the operand fetches could overlap to some extent

Comparison of MIPS/VAX running times

- Reasonable assumption: 2 ns cycle time, 10ns memory stall time
- If no cache, memory time dominates, VAX wins
- If everything in cache, no memory stalls, MIPS wins
- If cache is initially empty but large enough to contain everything, MIPS wins since it has better compute time

SPARC: A RISC Example

- Goals
  - Support object-oriented languages
  - Efficient trapping and subroutine call-return
  - RISC philosophy
- Fixed instruction size: 32 bits
- Small number of instruction formats (5 in V8)
- Load-store architecture, delayed branches
- 32 single-precision floating-point registers/16 double-precision
- 40 to 520 integer registers, BUT only 32 accessible at a time

Registers

- 32 registers visible at one time: `%r0` to `%r31`
- Four 8-register banks: inputs, outputs, globals, and locals
- Globals `%g0` to `%g7` correspond to `%r0` to `%r7`
- Register `%r0`/`%g0` always has value 0
- Outputs `%o0` to `%o7` correspond to `%r8` to `%r15`
- Locals `%l0` to `%l7` correspond to `%r16` to `%r23`
- Inputs `%i0` to `%i7` correspond to `%r24` to `%r31`

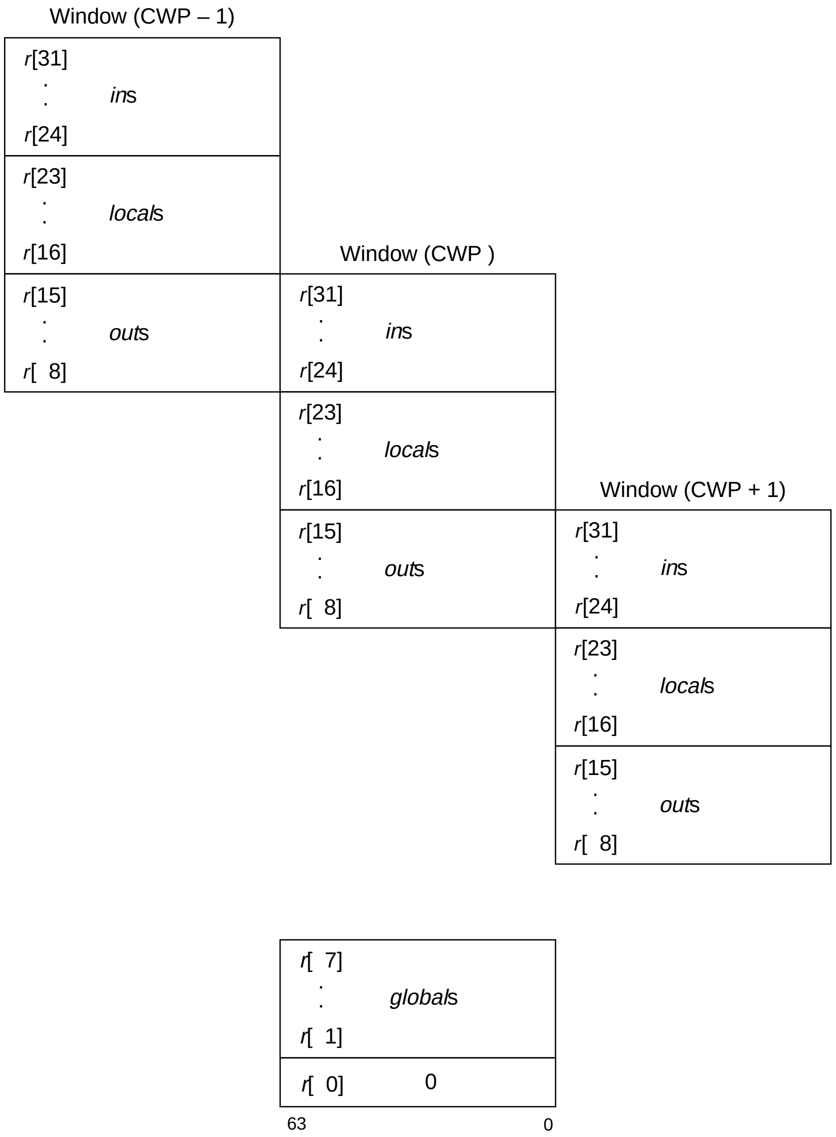
Registers

i7	*r*[31]
i6	*r*[30]
i5	*r*[29]
i4	*r*[28]
i3	*r*[27]
i2	*r*[26]
i1	*r*[25]
i0	*r*[24]
l7	*r*[23]
l6	*r*[22]
l5	*r*[21]
l4	*r*[20]
l3	*r*[19]
l2	*r*[18]
l1	*r*[17]
l0	*r*[16]
o7	*r*[15]
o6	*r*[14]
o5	*r*[13]
o4	*r*[12]
o3	*r*[11]
o2	*r*[10]
o1	*r*[9]
o0	*r*[8]
g7	*r*[7]
g6	*r*[6]
g5	*r*[5]
g4	*r*[4]
g3	*r*[3]
g2	*r*[2]
g1	*r*[1]
g0	*r*[0]

Register Window

- 32 registers visible at one time: `%r0` to `%r31`
- 40 to 520 registers in V8 implementation
- Inputs, Locals, and Outputs actually a 24-register *window* into master register file
- When call a subroutine, can shift register window forward so outputs become inputs, and get a new set of locals and outputs
- When return from a subroutine, can shift register window back
- Register "stack" managed as a circular buffer
- When overflows, trap to O/S to save off registers to stack in memory
- More efficient to save a lot of registers at once, and if have a lot of call-returns can avoid many register saves

Register Windows



Window (CWP – 1)

r[31] : *in*s *r*[24]

r[23] : *local*s *r*[16]

r[15] : *out*s *r*[8]

Window (CWP)

r[31] : *in*s *r*[24]

r[23] : *local*s *r*[16]

r[15] : *out*s *r*[8]

Window (CWP + 1)

r[31] : *in*s *r*[24]

r[23] : *local*s *r*[16]

r[15] : *out*s *r*[8]

r[7] : *global*s *r*[1]

r[0] 0

63 0

Register Window Management

- Register "stack" is a FIFO, managed as a circular array
- Each time we push or pop register file, modify current window index by 16
- When overflows, trap to O/S to save off registers to stack in memory
- More efficient to save a lot of registers at once
- Typically can avoid many register saves altogether
- Gives more efficient subroutine calls and interrupt/trap processing
- Good for real-time systems and certain programming languages (FORTH, Smalltalk, LISP, C++)

SPARC Instruction Format

Format 1 (op=1) CALL

op	disp30
31 30	29 0

Format 2 (op=0) SETHI, Branches

op	rd	op2	imm22
31 30	29 25	24 22	21 0

op	a	cond	op2	disp2
31 30	29	28 25	24 22	21 0

Format 3 (op=2 or 3) Arithmetic, Logical, Load, Store

op	rd	op3	rs1	i=0	---	rs2
31 30	29 25	24 19	18 14	13	12 5	4 0

op	rd	op3	rs1	i=1	simm13
31 30	29 25	24 19	18 14	13	12 0

op	rd	op3	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 12 5	4 0

Intel Pentium: Basic Instruction Set

The instruction set is very large

- 48 move/data instructions
- 12 binary arithmetic instruction
  ADD, ADC (add w/carry), SUB, MUL, DIV, INC, CMP
- 6 decimal arithmetic instructions
- 4 logic instructions (AND, OR, XOR, NOT)
- 10 shift/rotate instructions
- 48 control transfer instructions

Instruction Set (cont)

- 37 bit/byte instructions
  Testing bits/bytes
- 57 string instructions
- 13 flag control instructions
  Set/clear conditional flags used by branches
- 5 segment register instructions
- 6 misc instructions (including NOP)

Additional floating point and SIMD (SSE) instructions

Addressing Modes

- 16-bit, 32-bit (mostly 32 bit)
- Segment-relative addresses

  SEGMENT + BASE + (INDEX*SCALE) + DISPLACE

Mode	Example
Register	MOV DS,AZ
Immediate	MOV AX,1000H
Direct Addressing	MOV [2000H],AX
Register Indirect	MOV DL, [SI]
Base Addressing	MOV AX,[BX+4]
Indexed Addressing	MOV [DI-8],RL
Based Indexed Addressing	MOV [GP][SI],AH
Based Indexed with Displacement	MOV CL,[BX+DI+2040H]

IA-32 Evolution

32-bit Pentium evolution starts with 80386

- 1985: 80386

  32-bit addressing
- 1989: 80486
  - Internal cache,
  - Instruction pipeline,
  - Integrated math coprocessor
- 1993: Pentium
  - Superscalar microarchitecture

    executes multiple instructions in parallel;
  - Split internal cache (L1)

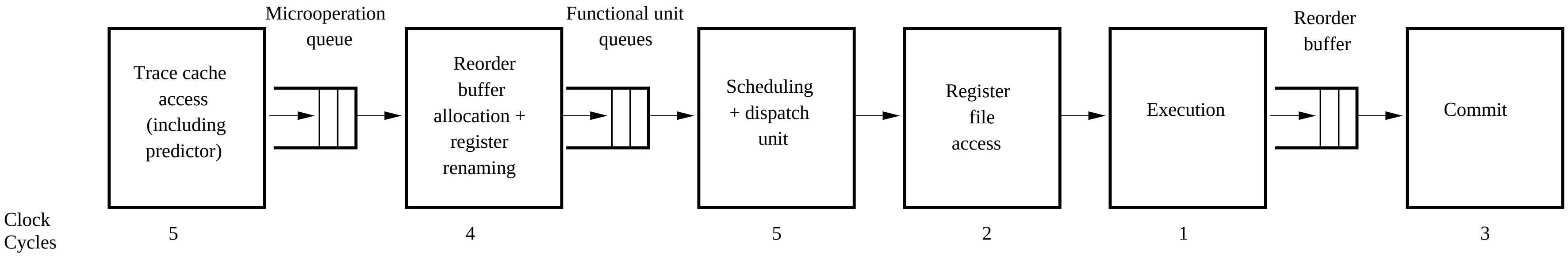
    separate instruction and data caches

- 1995: Pentium Pro
  - Backside level 2 cache (L2)
  - Convert complex instructions into micro-ops
    micro-ops execute on RISC
  - Longer pipeline

Copyright figure removed

- 1997: Pentium MMX, Pentium II
  - MMX instructions (integer SIMD instructions)
- 1999: Pentium III
  - SSE: SIMD single precision floats
    Useful for 3D graphics
  - 10 stage pipeline

- 2000: Pentium 4
  - SSE2: double precision floats, quadword ints, 128-bit ints
  - 20 stage pipeline
  - Branch prediction (4K branch predictor table)
  - Looks ahead 126 instructions to find parallel tasks
  - 12K Instruction cache
    stores micro-ops instead of instructions
  - L1: 8-KB, 4-way set associative, 64-byte lines
  - L2: 256KB (512KB), 8-way set associative, 128-byte lines
  - ALU executes common integer ops in one-half clock cycle


Microoperation queue
Functional unit queues
Reorder buffer
Trace cache access (including predictor)
Reorder buffer allocation + register renaming
Scheduling + dispatch unit
Register file access
Execution
Commit
Clock Cycles
5
4
5
2
1
3

- 2003,4: AMD, Intel announce extensions to 64 bit architectures

  SSE3 (graphics, video, thread sychronization support).

Where are we now?

- Power wall

Copyright figure removed

- Parallelism
  GPUs
- Smaller is better (for home market)
- Network applications

Take Aways

- Understanding hardware can improve compilers
  code rearrangement, loop unrolling
- Understanding hardware/OS can improve code
  Cache, virtual memory, paging
- Introduction to material you'll see later
  - Floating point
  - OS issues
  - Data structures
  - Interrupts, exceptions
  - Synchronization, parallelism

Example: Revisited

```
#include<stdio.h>
#define NR 10000
#define NC 10000

int a[NR][NC];

void main() {
  int i,j;
  for (i=0;i<NR;i++){
    for (j=0;j<NC;j++){
      a[i][j]=32767; } }
```

- Row-by-row (a[i][j]): 1.693 sec
- By column (a[j][i]): 27.045 sec
  (approx 16 times slower!)