Chapter 1

Introduction

- Rapidly changing field:
  - vacuum tube -> transistor -> IC -> VLSI (see section 1.4)
  - doubling every 1.5 years:
    memory capacity
    processor speed *(Due to advances in technology and organization)*
- Things you'll be learning:
  - how computers work, a basic foundation
  - how to analyze their performance (or how not to!)
  - issues affecting modern processors (caches, pipelines)
- Why learn this stuff?
  - you want to call yourself a "computer scientist"
  - you want to build software people use (need performance)
  - you need to make a purchasing decision or offer "expert" advice

What is a computer?

- Components:
  - input (mouse, keyboard)
  - output (display, printer)
  - memory (disk drives, DRAM, SRAM, CD)
  - network
- Our primary focus: the processor (datapath and control)
  - implemented using millions of transistors
  - Impossible to understand by looking at each transistor
  - We need...

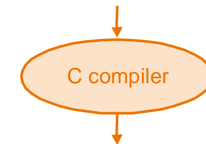
Abstraction

- **Delving into the depths reveals more information**
- **An abstraction omits unneeded detail, helps us cope with complexity**

What are some of the details that appear in these familiar abstractions?

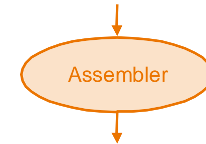
High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```



Assembly language program (for MIPS)

```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```



Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000100011100001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Instruction Set Architecture

- **A very important abstraction**
  - **interface between hardware and low-level software**
  - **standardizes instructions, machine language bit patterns, etc.**
  - **advantage:** ***different implementations of the same architecture***
  - **disadvantage:** ***sometimes prevents using new innovations***

True or False: Binary compatibility is extraordinarily important?

- **Modern instruction set architectures:**
  - **80x86/Pentium/K6, PowerPC, DEC Alpha, MIPS, SPARC, HP**

Where we are headed

- **Performance issues (Chapter 2)** ***vocabulary and motivation***
- **A specific instruction set architecture (Chapter 3)**
- **Arithmetic and how to build an ALU (Chapter 4)**
- **Constructing a processor to execute our instructions (Chapter 5)**
- **Pipelining to improve performance (Chapter 6)**
- **Memory: caches and virtual memory (Chapter 7)**
- **I/O (Chapter 8)**

Key to a good grade: reading the book!

Chapter 2

Performance

- **Measure, Report, and Summarize**
- **Make intelligent choices**
- **See through the marketing hype**
- **Key to understanding underlying organizational motivation**

Why is some hardware better than others for different programs?

What factors of system performance are hardware related?
(e.g., Do we need a new machine, or a new operating system?)

How does the machine's instruction set affect performance?

Which of these airplanes has the best performance?

Airplane	Passengers	Range (mi)	Speed (mph)
Boeing 737-100	101	630	598
Boeing 747	470	4150	610
BAC/Sud Concorde	132	4000	1350
Douglas DC-8-50	146	8720	544

- **How much faster is the Concorde compared to the 747?**
- **How much bigger is the 747 than the Douglas DC-8?**

Computer Performance: TIME, TIME, TIME

- **Response Time (latency)**
  - **— How long does it take for my job to run?**
  - **— How long does it take to execute a job?**
  - **— How long must I wait for the database query?**
- **Throughput**
  - **— How many jobs can the machine run at once?**
  - **— What is the average execution rate?**
  - **— How much work is getting done?**

- ***If we upgrade a machine with a new processor what do we increase?***

If we add a new machine to the lab what do we increase?

Execution Time

- **Elapsed Time**
  - **counts everything** ***(disk and memory accesses, I/O , etc.)***
  - **a useful number, but often not good for comparison purposes**
- **CPU time**
  - **doesn't count I/O or time spent running other programs**
  - **can be broken up into system time, and user time**

- **Our focus: user CPU time**
  - **time spent executing the lines of code that are "in" our program**

Book's Definition of Performance

- **For some program running on machine X,**

  $\text{Performance}_X = 1 / \text{Execution time}_X$

- **"X is n times faster than Y"**

  $\text{Performance}_X / \text{Performance}_Y = n$

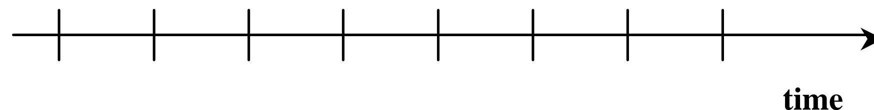
- **Problem:**
  - **machine A runs a program in 20 seconds**
  - **machine B runs the same program in 25 seconds**

Clock Cycles

- **Instead of reporting execution time in seconds, we often use cycles**

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

- **Clock "ticks" indicate when to start activities (one abstraction):**



- **cycle time = time between ticks = seconds per cycle**
- **clock rate (frequency) = cycles per second (1 Hz. = 1 cycle/sec)**

A 200 Mhz. clock has a $\frac{1}{200 \times 10^6} \times 10^9 = 5 \text{ nanoseconds}$ **cycle time**

How to Improve Performance

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

So, to improve performance (everything else being equal) you can either

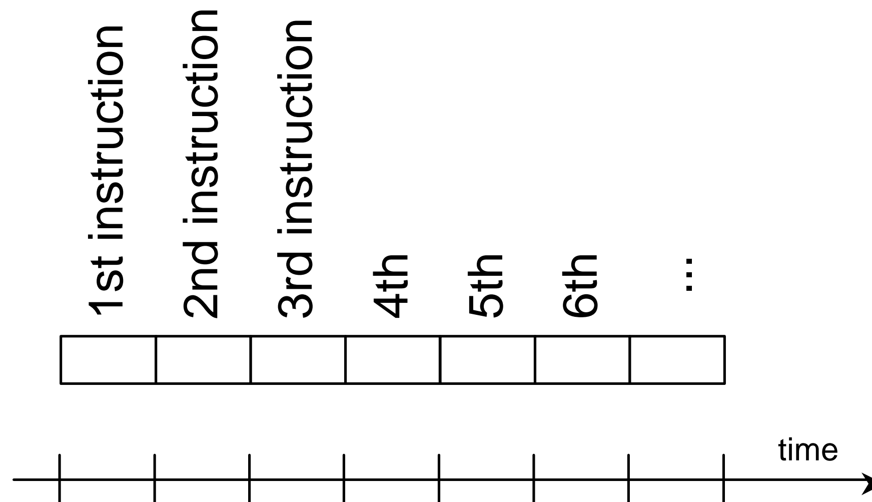
________ the # of required cycles for a program, or

________ the clock cycle time or, said another way,

________ the clock rate.

How many cycles are required for a program?

- **Could assume that # of cycles = # of instructions**

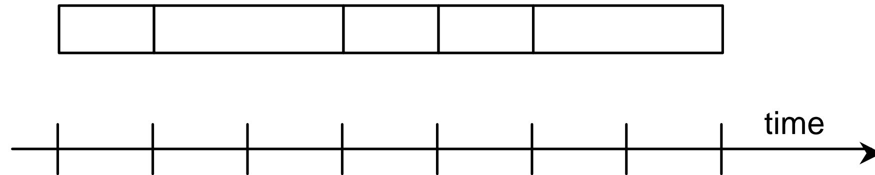


This assumption is incorrect,

different instructions take different amounts of time on different machines.

Why? hint: remember that these are machine instructions, not lines of C code

Different numbers of cycles for different instructions



- **Multiplication takes more time than addition**
- **Floating point operations take longer than integer ones**
- **Accessing memory takes more time than accessing registers**

- ***Important point: changing the cycle time often changes the number of cycles required for various instructions (more later)***

Example

- **Our favorite program runs in 10 seconds on computer A, which has a 400 Mhz. clock. We are trying to help a computer designer build a new machine B, that will run this program in 6 seconds. The designer can use new (or perhaps more expensive) technology to substantially increase the clock rate, but has informed us that this increase will affect the rest of the CPU design, causing machine B to require 1.2 times as many clock cycles as machine A for the same program. What clock rate should we tell the designer to target?"**

- **Don't Panic, can easily work this out from basic principles**

Now that we understand cycles

- **A given program will require**
  - **some number of instructions (machine instructions)**
  - **some number of cycles**
  - **some number of seconds**
- **We have a vocubulary that relates these quantities:**
  - **cycle time (seconds per cycle)**
  - **clock rate (cycles per second)**
  - **CPI (cycles per instruction)**

    a floating point intensive application might have a higher CPI
  - **MIPS (millions of instructions per second)**

    this would be higher for a program using simple instructions

Performance

- Performance is determined by execution time
- Do any of the other variables equal performance?
  - # of cycles to execute program?
  - # of instructions in program?
  - # of cycles per second?
  - average # of cycles per instruction?
  - average # of instructions per second?

- Common pitfall: thinking one of the variables is indicative of performance when it really isn't.

CPI Example

- **Suppose we have two implementations of the same instruction set architecture (ISA).**

  For some program,

  Machine A has a clock cycle time of 10 ns. and a CPI of 2.0
  Machine B has a clock cycle time of 20 ns. and a CPI of 1.2

  What machine is faster for this program, and by how much?

- ***If two machines have the same ISA which of our quantities (e.g., clock rate, CPI, execution time, # of instructions, MIPS) will always be identical?***

of Instructions Example

- A compiler designer is trying to decide between two code sequences for a particular machine. Based on the hardware implementation, there are three different classes of instructions: Class A, Class B, and Class C, and they require one, two, and three cycles (respectively).

  The first code sequence has 5 instructions: 2 of A, 1 of B, and 2 of C
  The second sequence has 6 instructions: 4 of A, 1 of B, and 1 of C.

  Which sequence will be faster? How much?
  What is the CPI for each sequence?

MIPS example

- **Two different compilers are being tested for a 100 MHz. machine with three different classes of instructions: Class A, Class B, and Class C, which require one, two, and three cycles (respectively). Both compilers are used to produce code for a large piece of software.**

  The first compiler's code uses 5 million Class A instructions, 1 million Class B instructions, and 1 million Class C instructions.

  The second compiler's code uses 10 million Class A instructions, 1 million Class B instructions, and 1 million Class C instructions.

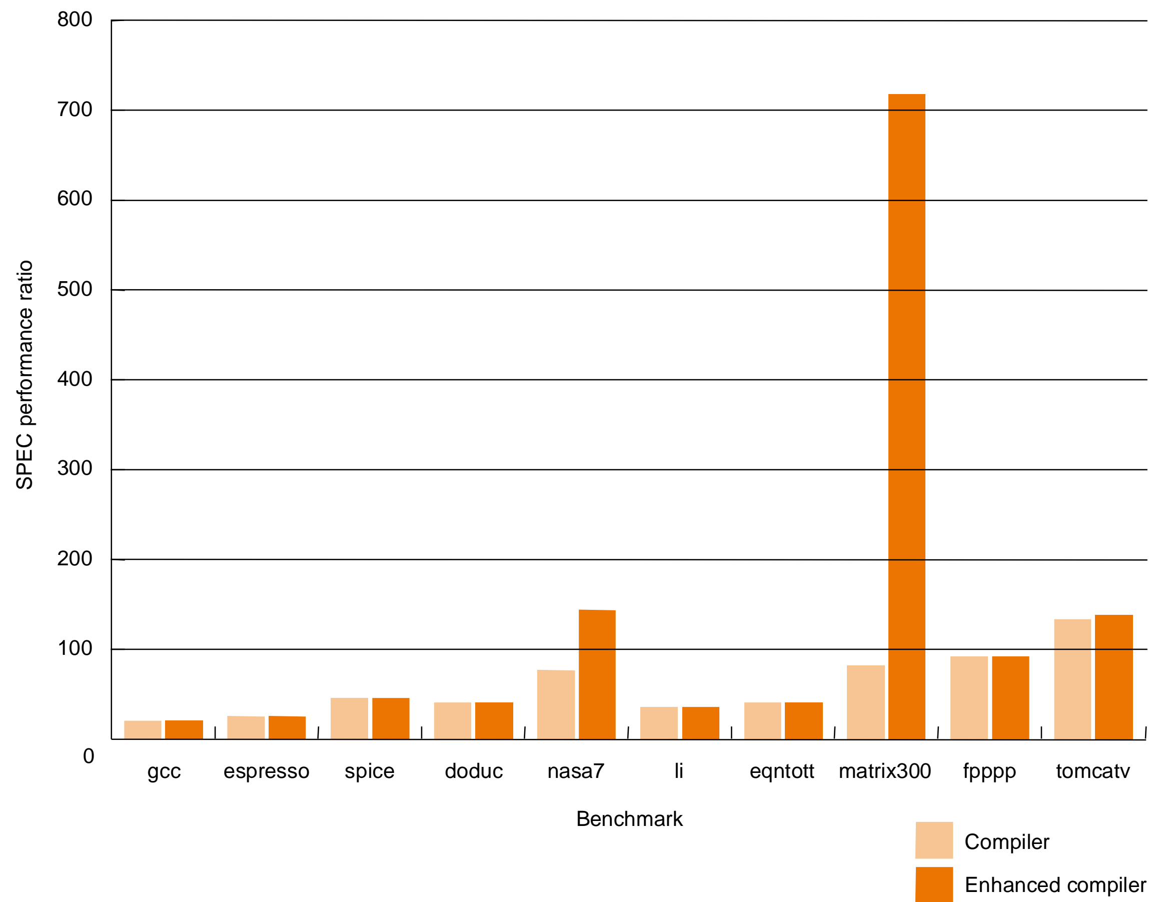
- **Which sequence will be faster according to MIPS?**
- **Which sequence will be faster according to execution time?**

Benchmarks

- Performance best determined by running a real application
  - Use programs typical of expected workload
  - Or, typical of expected class of applications
    e.g., compilers/editors, scientific applications, graphics, etc.
- Small benchmarks
  - nice for architects and designers
  - easy to standardize
  - can be abused
- SPEC (System Performance Evaluation Cooperative)
  - companies have agreed on a set of real program and inputs
  - can still be abused (Intel's "other" bug)
  - valuable indicator of performance (and compiler technology)

SPEC '89

- **Compiler "enhancements" and performance**



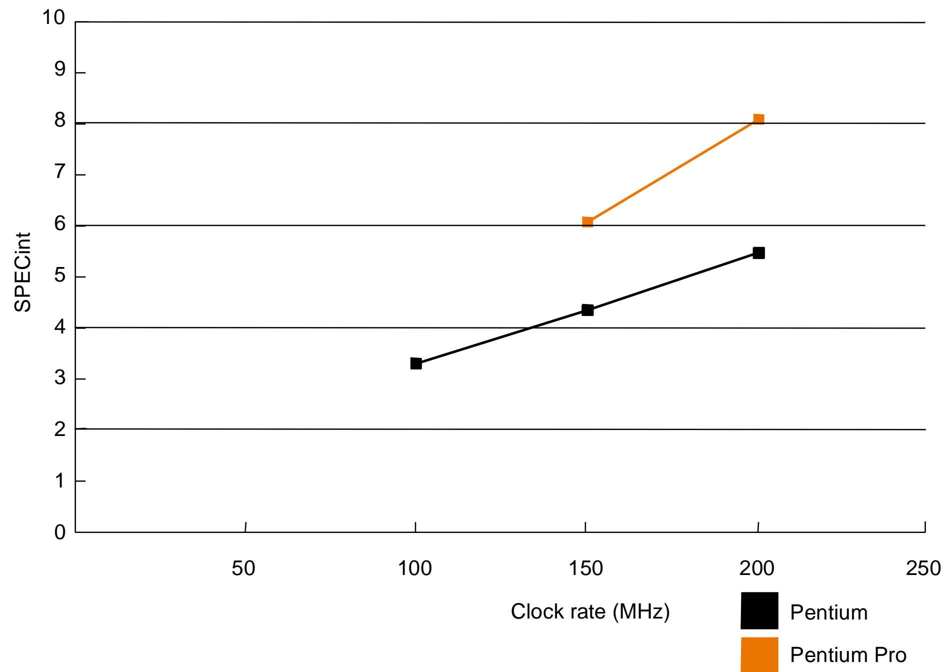
SPEC '95

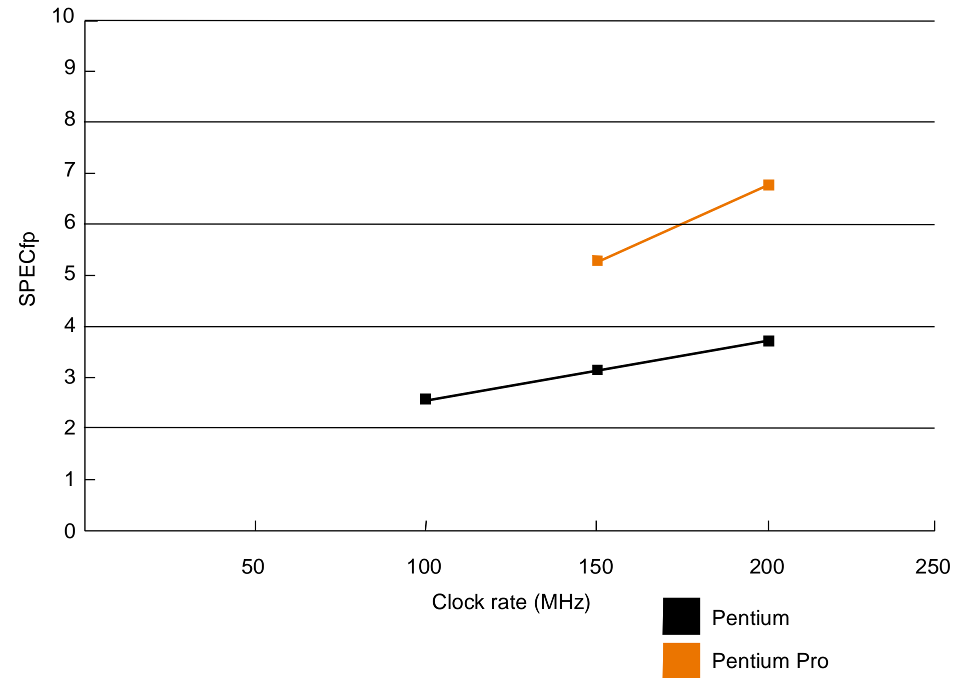
Benchmark	Description
go	Artificial intelligence; plays the game of Go
m88ksim	Motorola 88k chip simulator; runs test program
gcc	The Gnu C compiler generating SPARC code
compress	Compresses and decompresses file in memory
li	Lisp interpreter
ijpeg	Graphic compression and decompression
perl	Manipulates strings and prime numbers in the special-purpose programming language Perl
vortex	A database program
tomcatv	A mesh generation program
swim	Shallow water model with 513 x 513 grid
su2cor	quantum physics; Monte Carlo simulation
hydro2d	Astrophysics; Hydrodynamic Naiver Stokes equations
mgrid	Multigrid solver in 3-D potential field
applu	Parabolic/elliptic partial differential equations
trub3d	Simulates isotropic, homogeneous turbulence in a cube
apsi	Solves problems regarding temperature, wind velocity, and distribution of pollutant
fpppp	Quantum chemistry
wave5	Plasma physics; electromagnetic particle simulation

SPEC '95

Does doubling the clock rate double the performance?

Can a machine with a slower clock rate have better performance?





Amdahl's Law

Execution Time After Improvement =

Execution Time Unaffected +(Execution Time Affected / Amount of Improvement)

- **Example:**

  "Suppose a program runs in 100 seconds on a machine, with multiply responsible for 80 seconds of this time. How much do we have to improve the speed of multiplication if we want the program to run 4 times faster?"

  How about making it 5 times faster?

- *Principle: Make the common case fast*

Example

- **Suppose we enhance a machine making all floating-point instructions run five times faster. If the execution time of some benchmark before the floating-point enhancement is 10 seconds, what will the speedup be if half of the 10 seconds is spent executing floating-point instructions?**

- **We are looking for a benchmark to show off the new floating-point unit described above, and want the overall benchmark to show a speedup of 3. One benchmark we are considering runs for 100 seconds with the old floating-point hardware. How much of the execution time would floating-point instructions have to account for in this program in order to yield our desired speedup on this benchmark?**

Remember

- **Performance is specific to a particular program/s**
  - **Total execution time is a consistent summary of performance**

- **For a given architecture performance increases come from:**
  - **increases in clock rate (without adverse CPI affects)**
  - **improvements in processor organization that lower CPI**
  - **compiler enhancements that lower CPI and/or instruction count**

- **Pitfall: expecting improvement in one aspect of a machine's performance to affect the total performance**

- **You should not always believe everything you read! Read carefully! (see newspaper articles, e.g., Exercise 2.37)**

Chapter 3

Instructions:

- **Language of the Machine**
- **More primitive than higher level languages**
  e.g., no sophisticated control flow
- **Very restrictive**
  e.g., MIPS Arithmetic Instructions

- **We'll be working with the MIPS instruction set architecture**
  - **similar to other architectures developed since the 1980's**
  - **used by NEC, Nintendo, Silicon Graphics, Sony**

Design goals: maximize performance and minimize cost, reduce design time

MIPS arithmetic

- **All instructions have 3 operands**
- **Operand order is fixed (destination first)**

Example:

C code: `A = B + C`

MIPS code: `add $s0, $s1, $s2`

(associated with variables by compiler)

MIPS arithmetic

- Design Principle: simplicity favors regularity. Why?
- Of course this complicates some things...

C code:

```
A = B + C + D;
E = F - A;
```

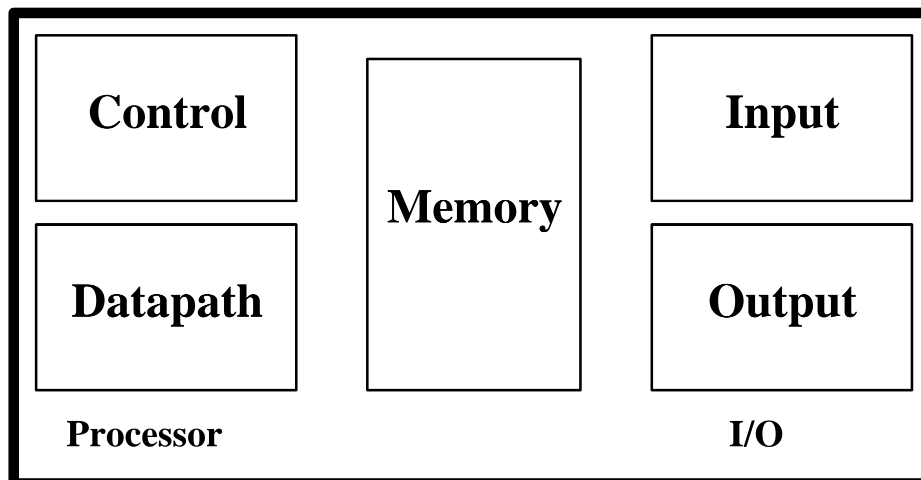
MIPS code:

```
add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0
```

- Operands must be registers, only 32 registers provided
- Design Principle: smaller is faster. Why?

Registers vs. Memory

- Arithmetic instructions operands must be registers,
  — only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables



Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory.

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data
...	

Memory Organization

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.

Address	Data
0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data
...	

Registers hold 32 bits of data

- 2^{32} bytes with byte addresses from 0 to $2^{32}-1$
- 2^{30} words with byte addresses 0, 4, 8, ... $2^{32}-4$
- Words are aligned

  i.e., what are the least 2 significant bits of a word address?

Instructions

- Load and store instructions
- Example:

C code:

```
A[8] = h + A[8];
```

MIPS code:

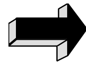
```
lw $t0, 32($s3)
add $t0, $s2, $t0
sw $t0, 32($s3)
```

- Store word has destination last
- Remember arithmetic operands are registers, not memory!

Our First Example

- **Can we figure out the code?**

```
swap(int v[], int k);
{ int temp;
    temp = v[k]
    v[k] = v[k+1];
    v[k+1] = temp;
}
```



```
swap:
    muli $2, $5, 4
    add $2, $4, $2
    lw $15, 0($2)
    lw $16, 4($2)
    sw $16, 0($2)
    sw $15, 4($2)
    jr $31
```

So far we've learned:

- **MIPS**
  - **— loading words but addressing bytes**
  - **— arithmetic on registers only**

- **Instruction** / **Meaning**

Instruction	Meaning
`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`
`sub $s1, $s2, $s3`	`$s1 = $s2 – $s3`
`lw $s1, 100($s2)`	`$s1 = Memory[$s2+100]`
`sw $s1, 100($s2)`	`Memory[$s2+100] = $s1`

Machine Language

- **Instructions, like registers and words of data, are also 32 bits long**
  - **Example:** `add $t0, $s1, $s2`
  - **registers have numbers,** `$t0=9, $s1=17, $s2=18`

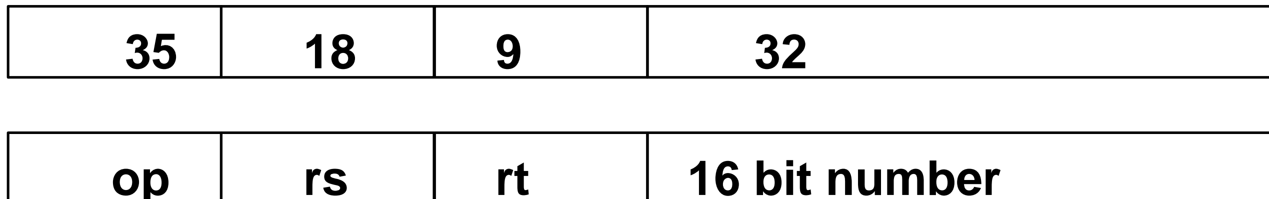
- **Instruction Format:**

000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct

- *Can you guess what the field names stand for?*

Machine Language

- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?
  - New principle: Good design demands a compromise
- Introduce a new type of instruction format
  - I-type for data transfer instructions
  - other format was R-type for register
- Example: `lw $t0, 32($s2)`

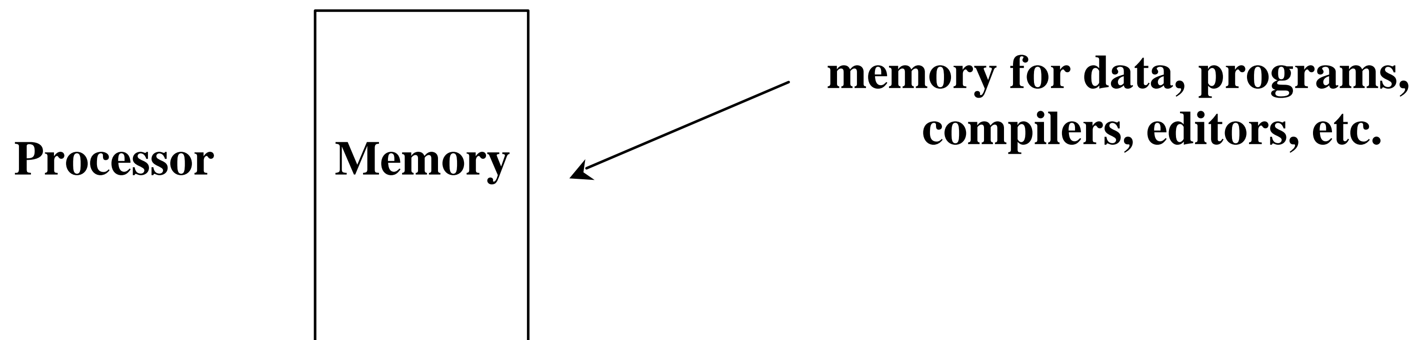


35	18	9	32
op	rs	rt	16 bit number

- Where's the compromise?

Stored Program Concept

- Instructions are bits
- Programs are stored in memory
  — to be read or written just like data



- Fetch & Execute Cycle
  - Instructions are fetched and put into a special register
  - Bits in the register "control" the subsequent actions
  - Fetch the “next” instruction and continue

Control

- **Decision making instructions**
  - **alter the control flow,**
  - **i.e., change the "next" instruction to be executed**

- **MIPS conditional branch instructions:**

```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

- **Example:** **if (i==j) h = i + j;**

```
        bne $s0, $s1, Label
        add $s3, $s0, $s1
Label: ....
```

Control

- **MIPS unconditional branch instructions:**
```
j  label
```
- **Example:**

```
if (i!=j)                beq $s4, $s5, Lab1
    h=i+j;               add $s3, $s4, $s5
else                     j Lab2
    h=i-j;               Lab1:  sub $s3, $s4, $s5
                         Lab2:  ...
```

- *Can you build a simple for loop?*

So far:

- **Instruction** / **Meaning**

```
add $s1,$s2,$s3    $s1 = $s2 + $s3
sub $s1,$s2,$s3    $s1 = $s2 – $s3
lw $s1,100($s2)    $s1 = Memory[$s2+100]
sw $s1,100($s2)    Memory[$s2+100] = $s1
bne $s4,$s5,L      Next instr. is at Label if $s4 ° $s5
beq $s4,$s5,L      Next instr. is at Label if $s4 = $s5
j Label            Next instr. is at Label
```

- **Formats:**

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

Control Flow

- **We have: beq, bne, what about Branch-if-less-than?**
- **New instruction:**

```
                                 if  $s1 < $s2 then
                                        $t0 = 1
     slt $t0, $s1, $s2           else
                                        $t0 = 0
```

- **Can use this instruction to build** `"blt $s1, $s2, Label"`
  — can now build general control structures
- **Note that the assembler needs a register to do this,**
  — there are policy of use conventions for registers

Policy of Use Conventions

Name	Register number	Usage
`$zero`	0	the constant value 0
`$v0-$v1`	2-3	values for results and expression evaluation
`$a0-$a3`	4-7	arguments
`$t0-$t7`	8-15	temporaries
`$s0-$s7`	16-23	saved
`$t8-$t9`	24-25	more temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return address

Constants

- Small constants are used quite frequently (50% of operands)
  e.g., A = A + 5;
  B = B + 1;
  C = C - 18;
- Solutions? Why not?
  - put 'typical constants' in memory and load them.
  - create hard-wired registers (like $zero) for constants like one.
- MIPS Instructions:

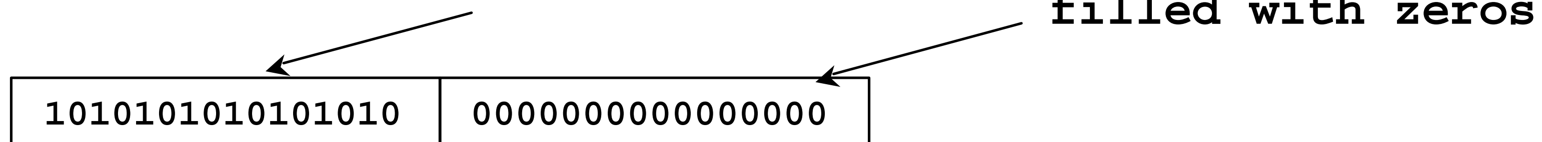
```
addi $29, $29, 4
slti $8, $18, 10
andi $29, $29, 6
ori $29, $29, 4
```

- How do we make this work?

How about larger constants?

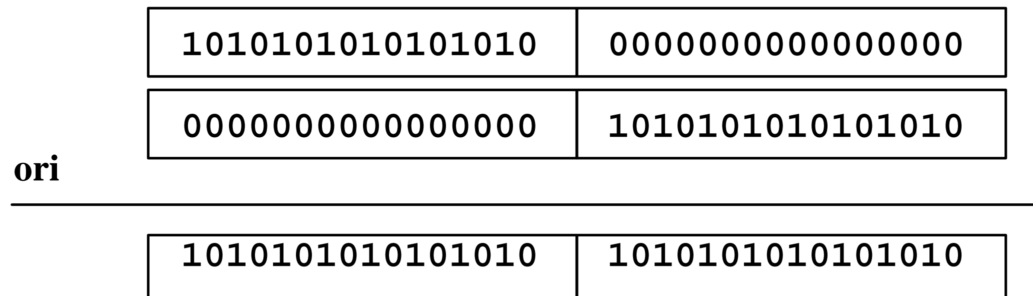
- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```



- Then must get the lower order bits right, i.e.,

```
ori $t0, $t0, 1010101010101010
```



Assembly Language vs. Machine Language

- Assembly provides convenient symbolic representation
  - much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
- Assembly can provide 'pseudoinstructions'
  - e.g., “move $t0, $t1” exists only in Assembly
  - would be implemented using “add $t0,$t1,$zero”
- When considering performance you should count real instructions

Other Issues

- Things we are not going to cover
  support for procedures
  linkers, loaders, memory layout
  stacks, frames, recursion
  manipulating strings and pointers
  interrupts and exceptions
  system calls and conventions
- Some of these we'll talk about later
- We've focused on architectural issues
  - basics of MIPS assembly language and machine code
  - we’ll build a processor to execute these instructions.

Overview of MIPS

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

- rely on compiler to achieve performance
  — what are the compiler's goals?
- help compiler where we can

Addresses in Branches and Jumps

- **Instructions:**

`bne $t4,$t5,Label`	Next instruction is at Label if `$t4 ° $t5`
`beq $t4,$t5,Label`	Next instruction is at Label if `$t4 = $t5`
`j Label`	Next instruction is at Label

- **Formats:**

I	op	rs	rt	16 bit address
J	op	26 bit address		

- **Addresses are not 32 bits**
  - **— How do we handle this with load and store instructions?**

Addresses in Branches

- **Instructions:**

  `bne $t4,$t5,Label` **Next instruction is at Label if $t4°$t5**
  `beq $t4,$t5,Label` **Next instruction is at Label if $t4=$t5**

- **Formats:**

- **Could specify a register (like lw and sw) and add it to address**
  - **use Instruction Address Register (PC = program counter)**
  - **most branches are local (principle of locality)**
- **Jump instructions just use high order bits of PC**
  - **address boundaries of 256 MB**

To summarize:

MIPS operands

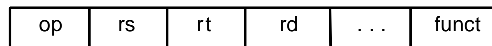
Name	Example	Comments
32 registers	$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

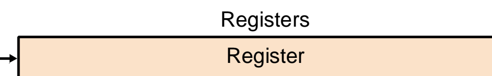
MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1, $s2, $s3	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	sub $s1, $s2, $s3	$s1 = $s2 - $s3	Three operands; data in registers
	add immediate	addi $s1, $s2, 100	$s1 = $s2 + 100	Used to add constants
Data transfer	load word	lw $s1, 100($s2)	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	sw $s1, 100($s2)	Memory[$s2 + 100] = $s1	Word from register to memory
	load byte	lb $s1, 100($s2)	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	sb $s1, 100($s2)	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	lui $s1, 100	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	beq $s1, $s2, 25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1, $s2, 25	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1, $s2, $s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set less than immediate	slti $s1, $s2, 100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

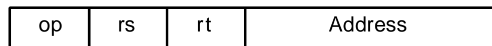
1. Immediate addressing

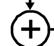
2. Register addressing

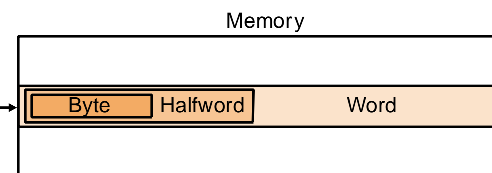




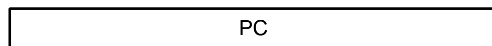
3. Base addressing

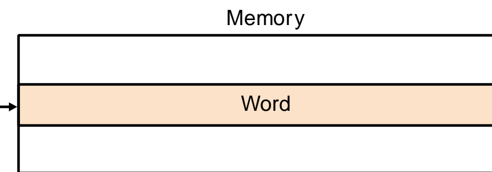




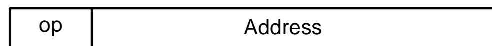


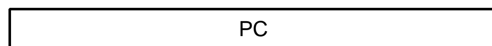
4. PC-relative addressing

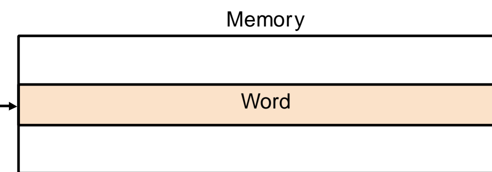




5. Pseudodirect addressing







Alternative Architectures

- Design alternative:
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI
- Sometimes referred to as “RISC vs. CISC”
  - virtually all new instruction sets since 1982 have been RISC
  - VAX: minimize code size, make assembly language easy

    instructions from 1 to 54 bytes long!
- We’ll look at PowerPC and 80x86

PowerPC

- **Indexed addressing**
  - **example:** `lw $t1,$a0+$s3  #$t1=Memory[$a0+$s3]`
  - **What do we have to do in MIPS?**
- **Update addressing**
  - **update a register as part of load (for marching through arrays)**
  - **example:** `lwu $t0,4($s3) #$t0=Memory[$s3+4];$s3=$s3+4`
  - **What do we have to do in MIPS?**
- **Others:**
  - **load multiple/store multiple**
  - **a special counter register** “`bc Loop`”

    decrement counter, if not 0 goto loop

80x86

- **1978: The Intel 8086 is announced (16 bit architecture)**
- **1980: The 8087 floating point coprocessor is added**
- **1982: The 80286 increases address space to 24 bits, +instructions**
- **1985: The 80386 extends to 32 bits, new addressing modes**
- **1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)**
- **1997: MMX is added**

"This history illustrates the impact of the "golden handcuffs" of compatibility

"adding new features as someone might add clothing to a packed bag"

"an architecture that is difficult to explain and impossible to love"

A dominant architecture: 80x86

- See your textbook for a more detailed description
- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
    e.g., “base or scaled index with 8 or 32 bit displacement”
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

“what the 80x86 lacks in style is made up in quantity,
making it beautiful from the right perspective”

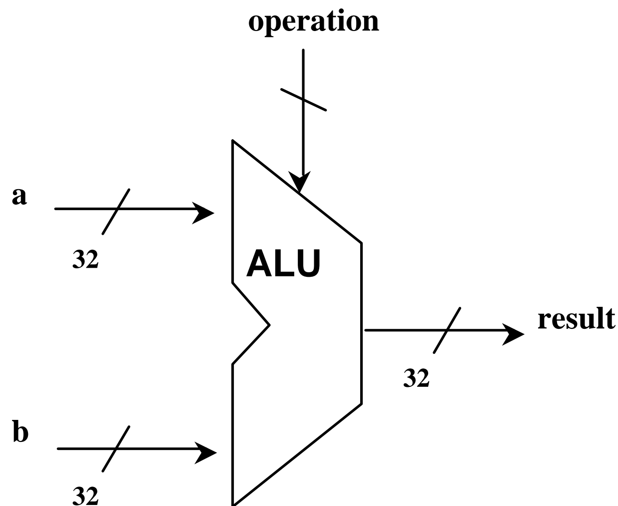
Summary

- Instruction complexity is only one variable
  - lower instruction count vs. higher CPI / lower clock rate
- Design Principles:
  - simplicity favors regularity
  - smaller is faster
  - good design demands compromise
  - make the common case fast
- Instruction set architecture
  - a very important abstraction indeed!

Chapter Four

Arithmetic

- Where we've been:
  - Performance (seconds, cycles, instructions)
  - Abstractions:
    Instruction Set Architecture
    Assembly Language and Machine Language
- What's up ahead:
  - Implementing the Architecture



Numbers

- **Bits are just bits (no inherent meaning)**
  — conventions define relationship between bits and numbers
- **Binary numbers (base 2)**
  0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  decimal: $0...2^n-1$
- **Of course it gets more complicated:**
  numbers are finite (overflow)
  fractions and real numbers
  negative numbers
  e.g., no MIPS subi instruction; addi can add a negative number)
- **How do we represent negative numbers?**
  i.e., which bit patterns will represent which numbers?

Possible Representations

Sign Magnitude:	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

- **Issues: balance, number of zeros, ease of operations**
- **Which one is best? Why?**

MIPS

- **32 bit signed numbers:**

```
0000 0000 0000 0000 0000 0000 0000 0000two = 0ten
0000 0000 0000 0000 0000 0000 0000 0001two = + 1ten
0000 0000 0000 0000 0000 0000 0000 0010two = + 2ten
...
0111 1111 1111 1111 1111 1111 1111 1110two = + 2,147,483,646ten
0111 1111 1111 1111 1111 1111 1111 1111two = + 2,147,483,647ten   maxint
1000 0000 0000 0000 0000 0000 0000 0000two = – 2,147,483,648ten   minint
1000 0000 0000 0000 0000 0000 0000 0001two = – 2,147,483,647ten
1000 0000 0000 0000 0000 0000 0000 0010two = – 2,147,483,646ten
...
1111 1111 1111 1111 1111 1111 1111 1101two = – 3ten
1111 1111 1111 1111 1111 1111 1111 1110two = – 2ten
1111 1111 1111 1111 1111 1111 1111 1111two = – 1ten
```

Two's Complement Operations

- **Negating a two's complement number: invert all bits and add 1**
  - **remember: “negate” and “invert” are quite different!**
- **Converting n bit numbers into numbers with more than n bits:**
  - **MIPS 16 bit immediate gets converted to 32 bits for arithmetic**
  - **copy the most significant bit (the sign bit) into the other bits**

```
0010  -> 0000 0010
1010  -> 1111 1010
```

  - **"sign extension"   (lbu vs. lb)**

Addition & Subtraction

- **Just like in grade school (carry/borrow 1s)**

```
  0111        0111        0110
+ 0110      - 0110      - 0101
```

- **Two's complement operations easy**
  - **subtraction using addition of negative numbers**

```
  0111
+ 1010
```

- **Overflow (result too large for finite computer word):**
  - **e.g., adding two n-bit numbers does not yield an n-bit number**

```
  0111
+ 0001
_ 1000
```

note that overflow term is somewhat misleading, it does not mean a carry "overflowed"

Detecting Overflow

- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction
- Overflow occurs when the value affects the sign:
  - overflow when adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive and get a negative
  - or, subtract a positive from a negative and get a positive
- Consider the operations A + B, and A – B
  - Can overflow occur if B is 0 ?
  - Can overflow occur if A is 0 ?

Effects of Overflow

- **An exception (interrupt) occurs**
  - **Control jumps to predefined address for exception**
  - **Interrupted address is saved for possible resumption**
- **Details based on software system / language**
  - **example: flight control vs. homework assignment**
- **Don't always want to detect overflow**
  — new MIPS instructions: `addu, addiu, subu`

note: `addiu` *still sign-extends!*
note: `sltu`, `sltiu` *for unsigned comparisons*

Review: Boolean Algebra & Gates

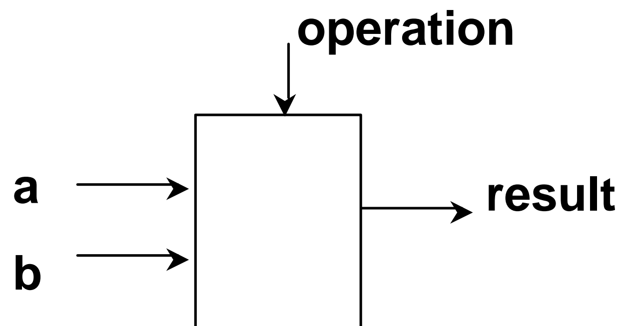
- Problem: Consider a logic function with three inputs: A, B, and C.

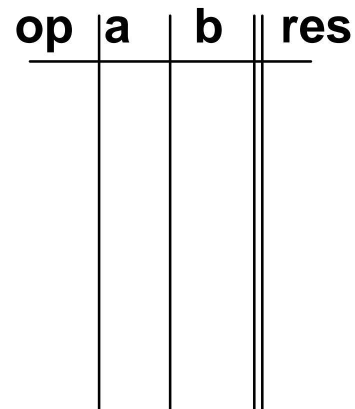
  Output D is true if at least one input is true
  Output E is true if exactly two inputs are true
  Output F is true only if all three inputs are true

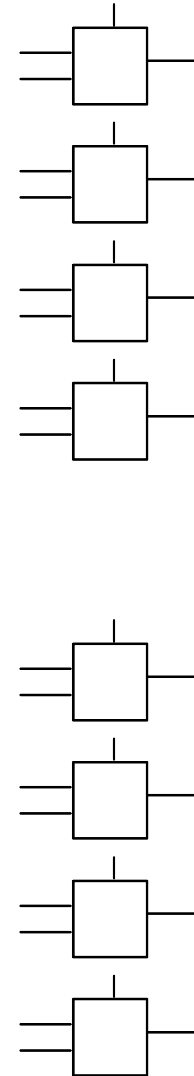
- Show the truth table for these three functions.
- Show the Boolean equations for these three functions.
- Show an implementation consisting of inverters, AND, and OR gates.

An ALU (arithmetic logic unit)

- **Let's build an ALU to support the `andi` and `ori` instructions**
  - **we'll just build a 1 bit ALU, and use 32 of them**



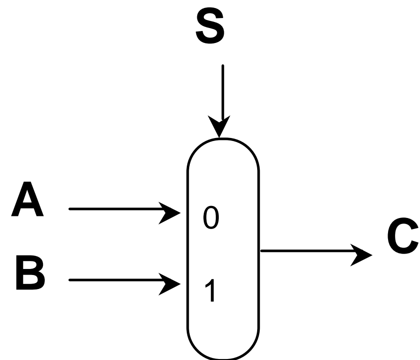




- **Possible Implementation (sum-of-products):**

Review: The Multiplexor

- Selects one of the inputs to be the output, based on a control input

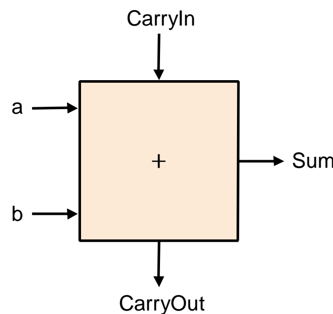


note: we call this a 2-input mux
even though it has 3 inputs!

- Lets build our ALU using a MUX:

Different Implementations

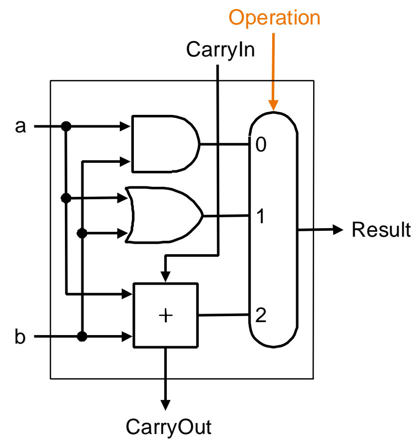
- Not easy to decide the “best” way to build something
  - Don't want too many inputs to a single gate
  - Dont want to have to go through too many gates
  - for our purposes, ease of comprehension is important
- Let's look at a 1-bit ALU for addition:

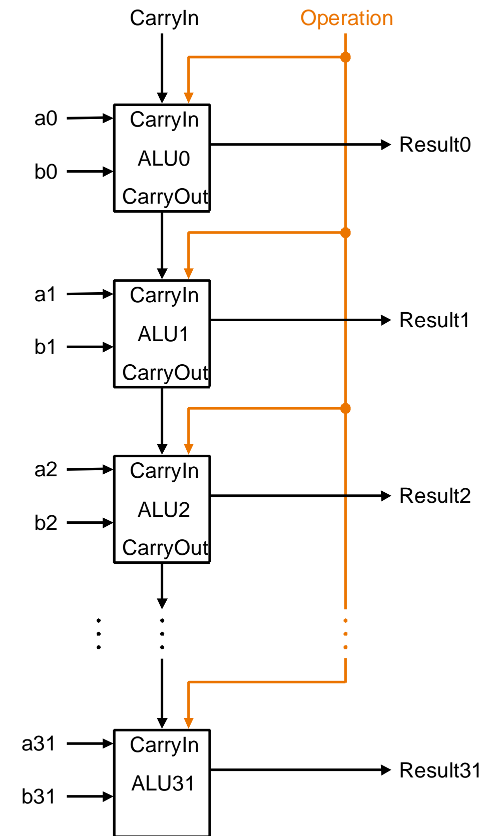


$$c_{out} = a\ b + a\ c_{in} + b\ c_{in}$$
$$sum = a \text{ xor } b \text{ xor } c_{in}$$

- How could we build a 1-bit ALU for add, and, and or?
- How could we build a 32-bit ALU?

Building a 32 bit ALU

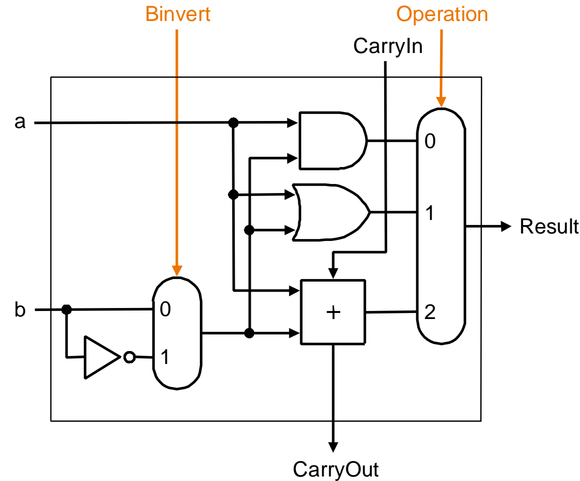




What about subtraction (a – b) ?

- Two's complement approch: just negate b and add.
- How do we negate?

- A very clever solution:

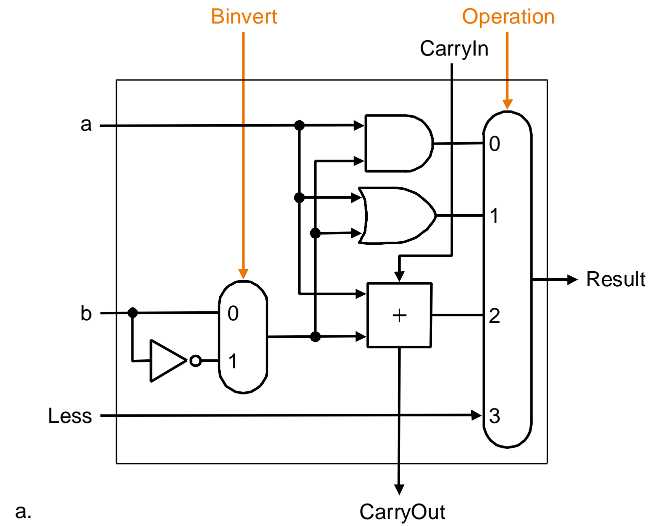


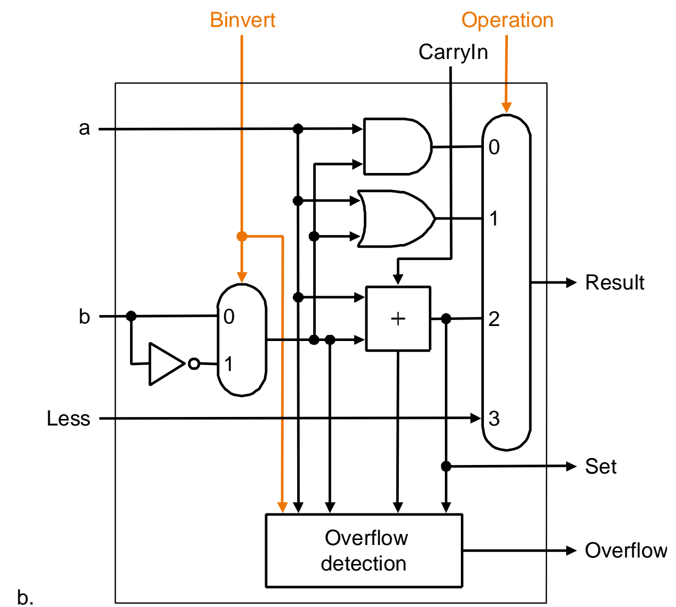
Tailoring the ALU to the MIPS

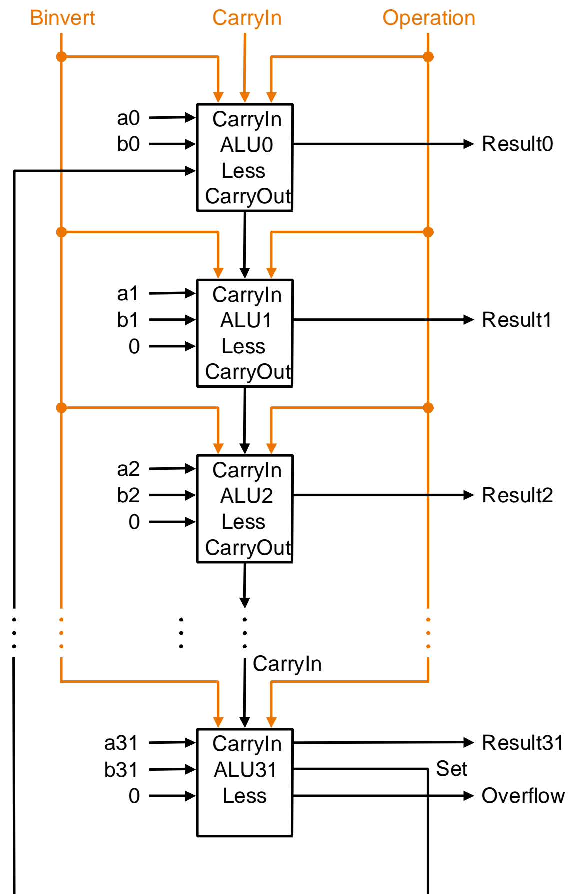
- **Need to support the set-on-less-than instruction (slt)**
  - **remember: slt is an arithmetic instruction**
  - **produces a 1 if rs < rt and 0 otherwise**
  - **use subtraction: (a-b) < 0 implies a < b**
- **Need to support test for equality (beq $t5, $t6, $t7)**
  - **use subtraction: (a-b) = 0 implies a = b**

Supporting slt

- **Can we figure out the idea?**





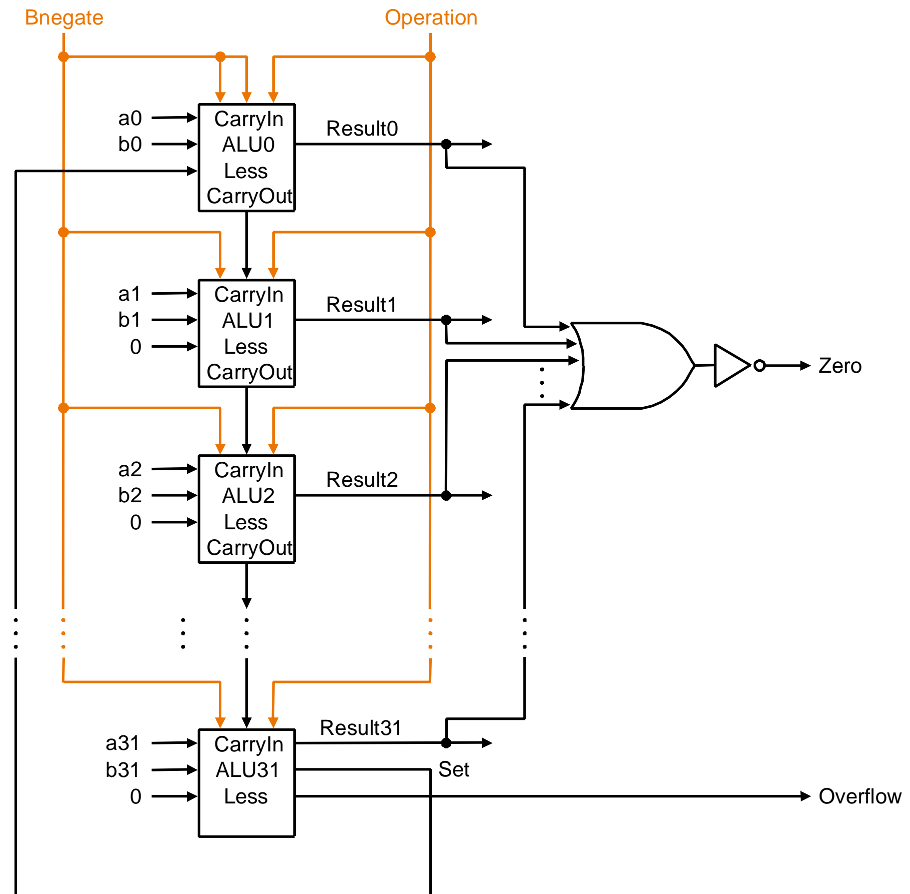

Binvert
CarryIn
Operation
a0
b0
CarryIn
ALU0
Less
CarryOut
Result0
a1
b1
0
CarryIn
ALU1
Less
CarryOut
Result1
a2
b2
0
CarryIn
ALU2
Less
CarryOut
Result2
CarryIn
a31
b31
0
CarryIn
ALU31
Less
Result31
Set
Overflow

Test for equality

- **Notice control lines:**

```
000 = and
001 = or
010 = add
110 = subtract
111 = slt
```

•*Note: zero is a 1 when the result is zero!*



Conclusion

- We can build an ALU to support the MIPS instruction set
  - key idea: use multiplexor to select the output we want
  - we can efficiently perform subtraction using two's complement
  - we can replicate a 1-bit ALU to produce a 32-bit ALU
- Important points about hardware
  - all of the gates are always working
  - the speed of a gate is affected by the number of inputs to the gate
  - the speed of a circuit is affected by the number of gates in series (on the "critical path" or the "deepest level of logic")
- Our primary focus: comprehension, however,
  - Clever changes to organization can improve performance (similar to using better algorithms in software)
  - we'll look at two examples for addition and multiplication

Problem: ripple carry adder is slow

- **Is a 32-bit ALU as fast as a 1-bit ALU?**
- **Is there more than one way to do addition?**
  - **two extremes: ripple carry and sum-of-products**

Can you see the ripple? How could you get rid of it?

$c_1 = b_0c_0 + a_0c_0 + a_0b_0$

$c_2 = b_1c_1 + a_1c_1 + a_1b_1c_2 =$

$c_3 = b_2c_2 + a_2c_2 + a_2b_2$ $\quad c_3 =$

$c_4 = b_3c_3 + a_3c_3 + a_3b_3$ $\quad c_4 =$

Not feasible! Why?

Carry-lookahead adder

- An approach in-between our two extremes
- Motivation:
  - If we didn't know the value of carry-in, what could we do?
  - When would we always generate a carry? $g_i = a_i b_i$
  - When would we propagate the carry? $p_i = a_i + b_i$
- Did we get rid of the ripple?

$c_1 = g_0 + p_0 c_0$

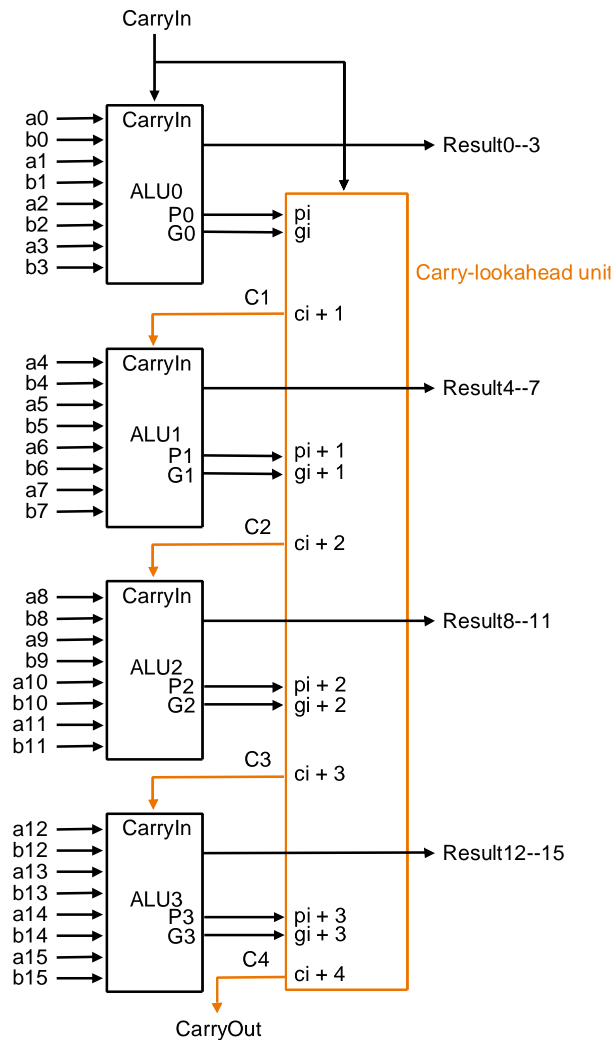
$c_2 = g_1 + p_1 c_1$ $c_2 =$

$c_3 = g_2 + p_2 c_2$ $c_3 =$

$c_4 = g_3 + p_3 c_3$ $c_4 =$

Feasible! Why?

Use principle to build bigger adders



- **Can't build a 16 bit adder this way... (too big)**
- **Could use ripple carry of 4-bit CLA adders**
- **Better: use the CLA principle again!**

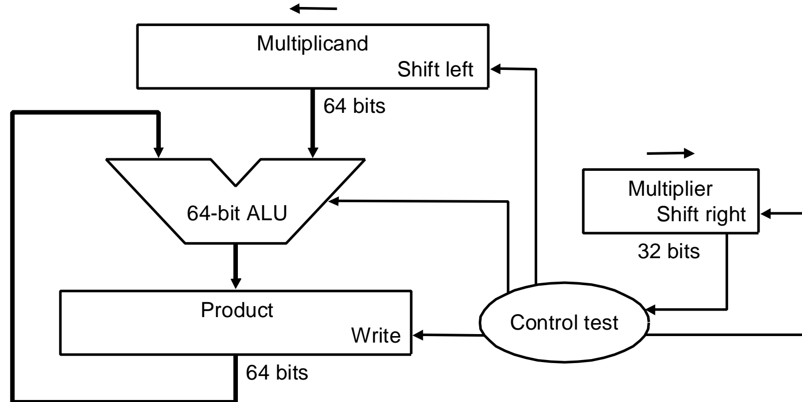
Multiplication

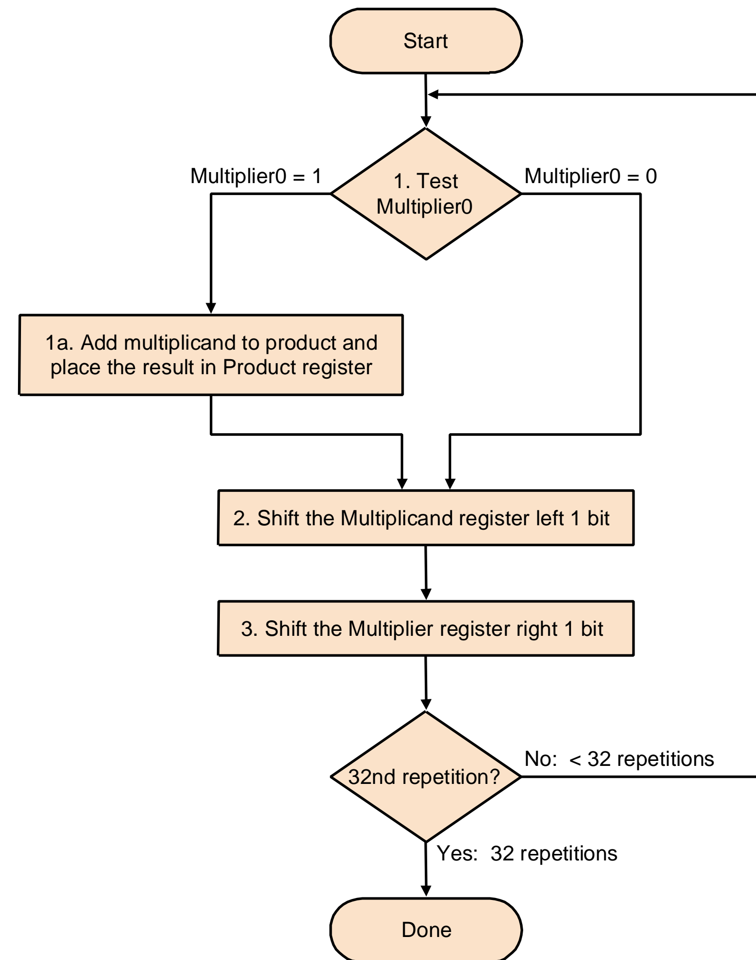
- More complicated than addition
  - accomplished via shifting and addition
- More time and more area
- Let's look at 3 versions based on gradeschool algorithm

```
     0010   (multiplicand)
___x_1011   (multiplier)
```

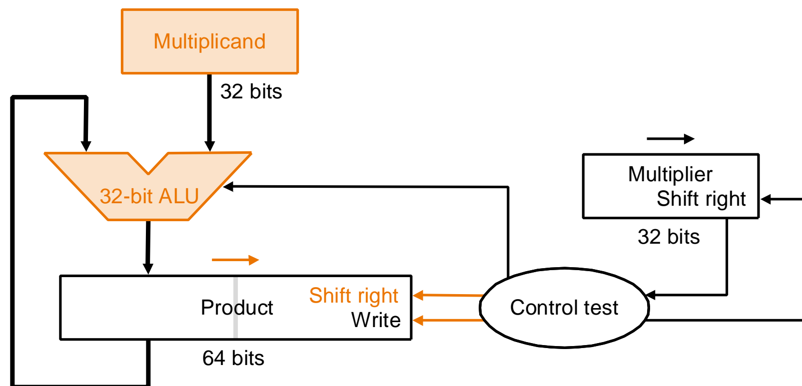
- Negative numbers: convert and multiply
  - there are better techniques, we won’t look at them

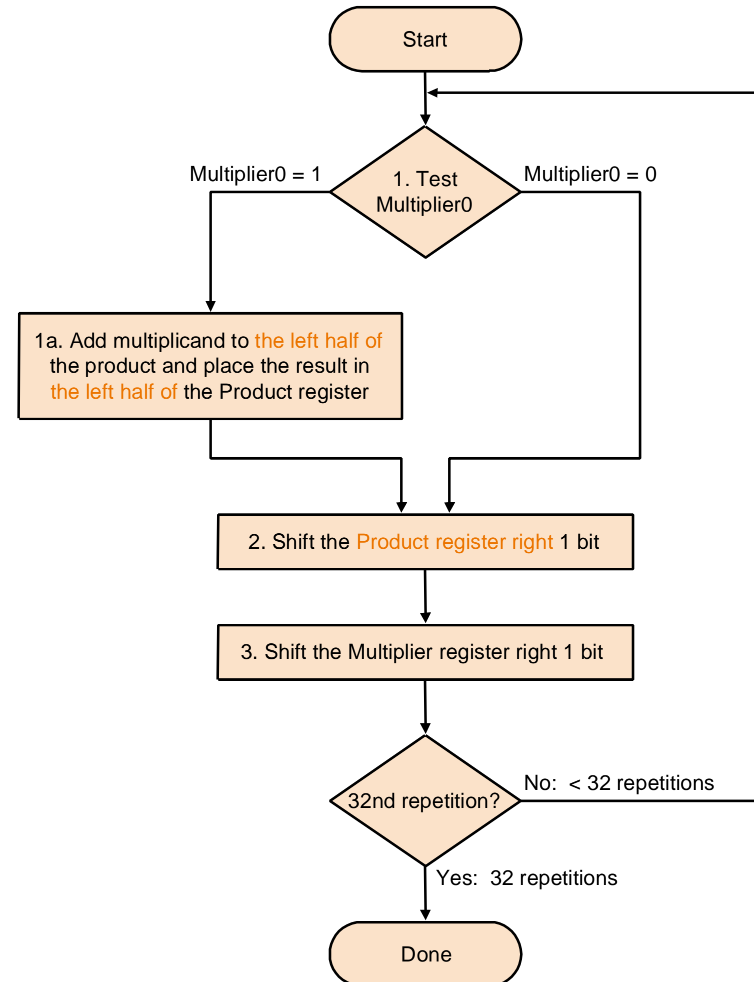
Multiplication: Implementation



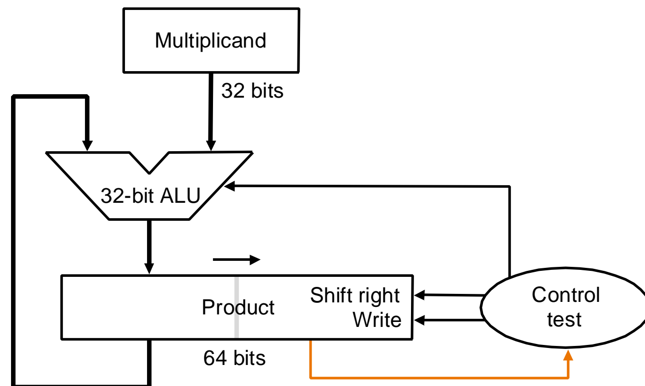


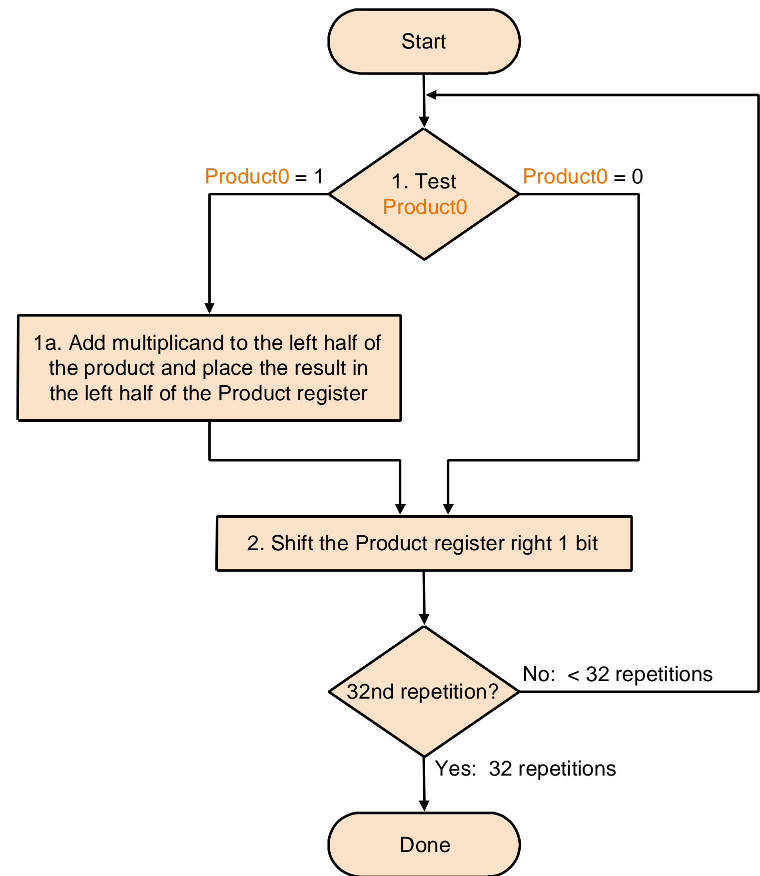
Second Version





Final Version





Floating Point (a brief look)

- We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., .000000001
  - very large numbers, e.g., 3.15576×10^9
- Representation:
  - sign, exponent, significand: $(-1)^{sign} \times significand \times 2^{exponent}$
  - more bits for significand gives more accuracy
  - more bits for exponent increases range
- IEEE 754 floating point standard:
  - single precision: 8 bit exponent, 23 bit significand
  - double precision: 11 bit exponent, 52 bit significand

IEEE 754 floating-point standard

- **Leading "1" bit of significand is implicit**

- **Exponent is "biased" to make sorting easier**
  - **all 0s is smallest exponent all 1s is largest**
  - **bias of 127 for single precision and 1023 for double precision**
  - **summary:** $(-1)^{sign} \times (1+\text{significand}) \times 2^{exponent - bias}$

- **Example:**

  - **decimal:** $-.75 = -3/4 = -3/2^2$
  - **binary:** $-.11 = -1.1 \times 2^{-1}$
  - **floating point: exponent = 126 = 01111110**

  - **IEEE single precision:** `10111111010000000000000000000000`

Floating Point Complexities

- Operations are somewhat more complicated (see text)
- In addition to overflow we can have “underflow”
- Accuracy can be a big problem
  - IEEE 754 keeps two extra bits, guard and round
  - four rounding modes
  - positive divided by zero yields “infinity”
  - zero divide by zero yields “not a number”
  - other complexities
- Implementing the standard can be tricky
- Not using the standard can be even worse
  - see text for description of 80x86 and Pentium bug!

Chapter Four Summary

- **Computer arithmetic is constrained by limited precision**
- **Bit patterns have no inherent meaning but standards do exist**
  - **two's complement**
  - **IEEE 754 floating point**
- **Computer instructions determine "meaning" of the bit patterns**
- **Performance and accuracy are important so there are many complexities in real machines (i.e., algorithms and implementation).**

- **We are ready to move on (and implement the processor)**

you may want to look back (Section 4.12 is great reading!)