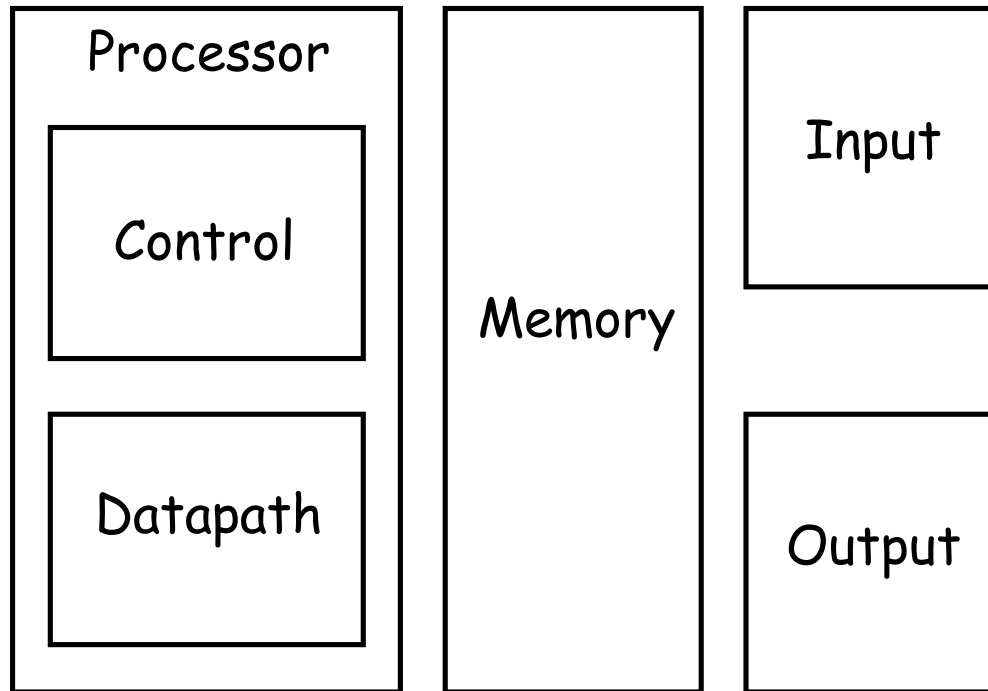


# About Computer Architecture

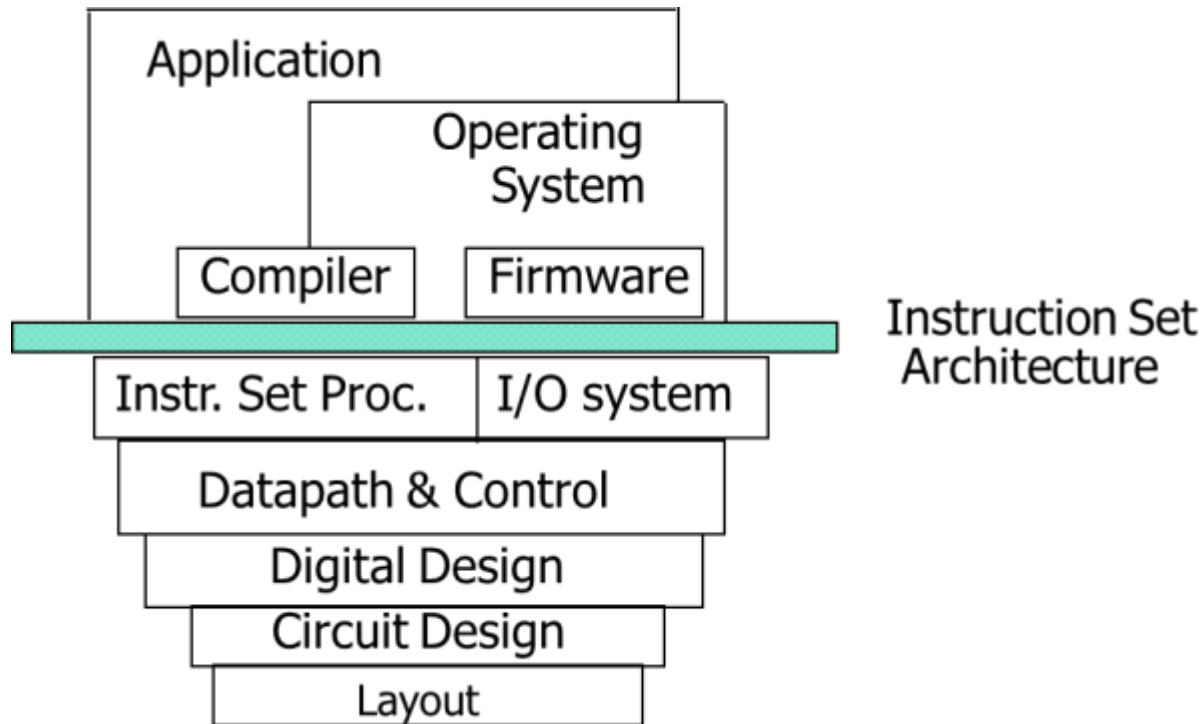
1. What ?
2. Why?
3. Where ?

# What?

- Since 1946 all computers have had 5 components

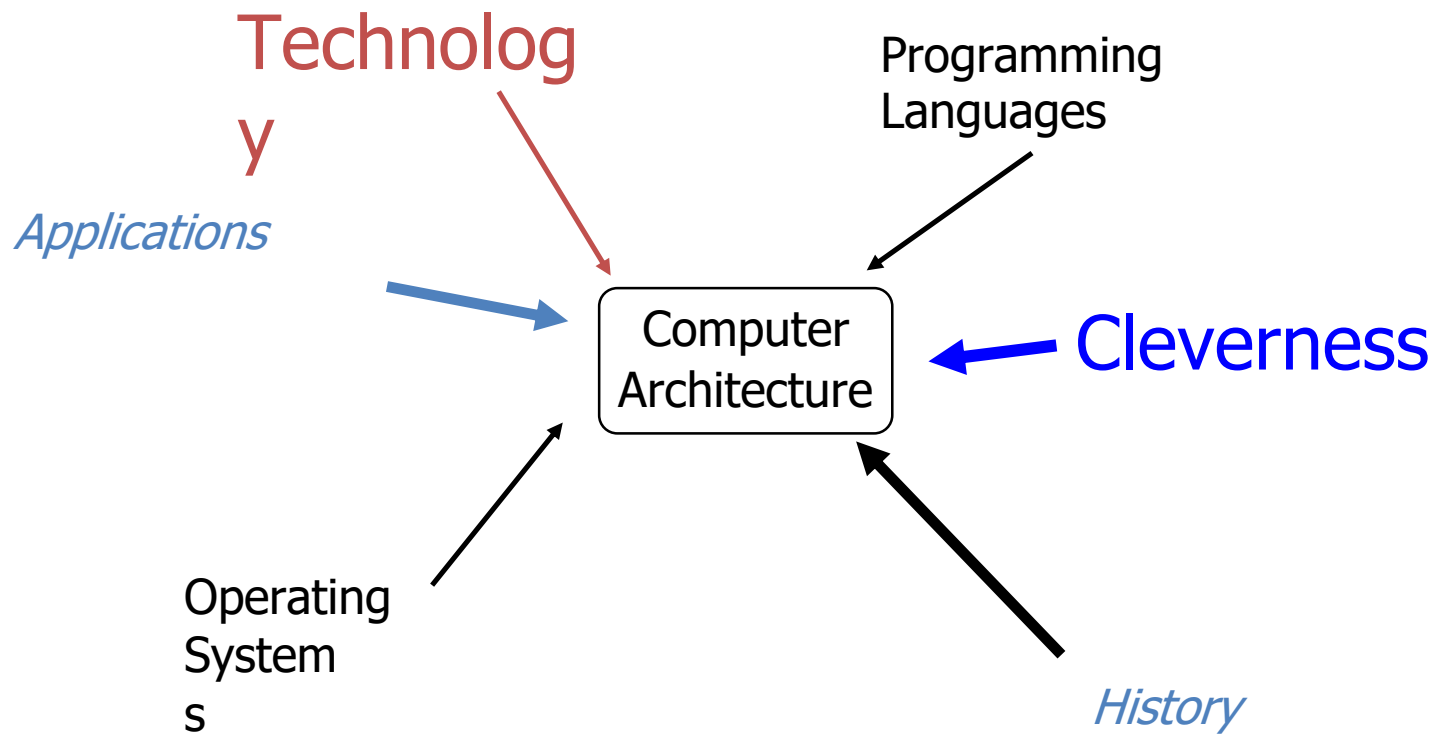


# Computer Architecture



- Coordination of many levels of abstraction
- Under a rapidly changing set of forces
- Design, Measurement, and Evaluation

# Why?



# Why Computer Organization

---

- Embarrassing if you are a BS in CS/CE and can't make sense of the following terms: DRAM, pipelining, cache hierarchies, I/O, virtual memory, ...
- Embarrassing if you are a BS in CS/CE and can't decide which processor to buy: (helps us reason about performance/power), ...
- Obvious first step for chip designers, compiler/OS writers
- Will knowledge of the hardware help you write better and more secure programs?

# What Does This Mean to a Programmer?

---

- Today, one can expect only a 20% annual improvement; the improvement is even lower if the program is not multi-threaded
  - A program needs many threads
  - The threads need efficient synchronization and communication
  - Data placement in the memory hierarchy is important
  - Accelerators should be used when possible

# Challenges for Hardware Designers

---

- Find efficient ways to
  - improve single-thread performance and energy
  - improve data sharing
  - boost programmer productivity
  - manage the memory system
  - build accelerators for important kernels
  - provide security

# The HW/SW Interface

---

Application software

---

Systems software  
(OS, compiler)

---

Hardware

$a[i] = b[i] + c;$

↓ Compiler

```
lw    $15, 0($2)
add   $16, $15, $14
add   $17, $15, $13
lw    $18, 0($12)
lw    $19, 0($17)
add   $20, $18, $19
sw    $20, 0($16)
```

↓ Assembler

```
000000101100000
110100000100010
...
```





## CSE 317 Lecture 2

---

# Computer performance



# The Computer Revolution

---

1. Progress in computer technology .
2. Makes novel applications feasible.
  - Computer in Automobile.
  - Cell phone
  - Worldwide web
  - Search engine .....etc.



# Performance

---

- *Performance is the key to understanding underlying motivation for the hardware and its organization*
- Measure, report, and summarize performance to enable users to
  - make intelligent choices
  - see through the marketing hype!
- *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?*



# The Role of Performance

---

Response Time

Throughput

Relative performance

Measuring Execution time

CPU time

CPU clocking ,instruction count and CPI



# What do we measure?

## Define performance....

| <u>Airplane</u><br><u>(mph)</u> | <u>Passengers</u> | <u>Range (mi)</u> | <u>Speed</u> |
|---------------------------------|-------------------|-------------------|--------------|
|---------------------------------|-------------------|-------------------|--------------|

|                  |     |      |      |
|------------------|-----|------|------|
| 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 Boeing 747 than the Douglas DC-8?
- *So which of these airplanes has the best performance?!*



# Computer Performance: TIME, TIME, TIME!!!

- *Response Time (elapsed time, latency):*
  - how long does it take for *my* job to run?
  - how long does it take to execute (start to finish) *my* job?
  - how long must *I* wait for the database query?} Individual user concerns...
- *Throughput:*
  - how *many* jobs can the machine run at once?
  - what is the *average* execution rate?
  - how *much* work is getting done?} Systems manager concerns...
- *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, waiting for I/O, running other programs, etc.*) from start to finish
  - a useful number, but often not good for comparison purposes
- elapsed time = CPU time + wait time (I/O, other programs, etc.)

- *CPU time*

- doesn't count waiting for I/O or time spent running other programs
- can be divided into *user CPU time* and *system CPU time* (OS calls)

CPU time = user CPU time + system CPU time

⇒ elapsed time = user CPU time + system CPU time + wait time

- Our focus: *user CPU time* (*CPU execution time* or, simply, *execution time*)

- time spent executing the lines of code that are *in our program*



# Definition of Performance

---

- For some program running on machine X:

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

This means that for two computers X and Y, if the performance of X is greater than the performance of Y,

We have

$$\text{Performance}_X > \text{Performance}_Y$$

$$1 / \text{execution of time}_X > 1 / \text{execution time of } Y$$

$$\text{Execution time}_Y > \text{Execution time}_X$$

*X is n times faster than Y* means:

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





# Example of Relative performance

---

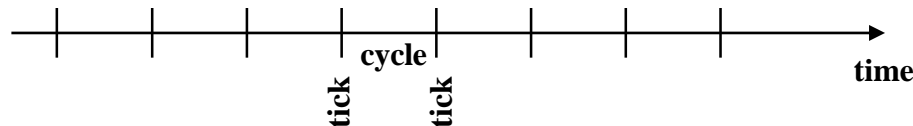
- If Afia's Computer runs a program in 10 seconds and Tahmid's computer runs the same program in 15 seconds, whose computer is faster and by how much faster?

# Clock Cycles

- Instead of reporting execution time in seconds, we often use *cycles*. In modern computers hardware events progress cycle by cycle: in other words, each event, e.g., multiplication, addition, etc., is a sequence of cycles

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

- Clock ticks* indicate start and end of cycles:



- cycle time* = time between ticks = seconds per cycle
- clock rate (frequency)* = cycles per second (1 Hz. = 1 cycle/sec, 1 MHz. =  $10^6$  cycles/sec)  $\frac{1}{200 \times 10^6} \times 10^9 = 5$  nanoseconds
- Example:* A 200 Mhz. clock has a cycle time



# Performance Equation I

---

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

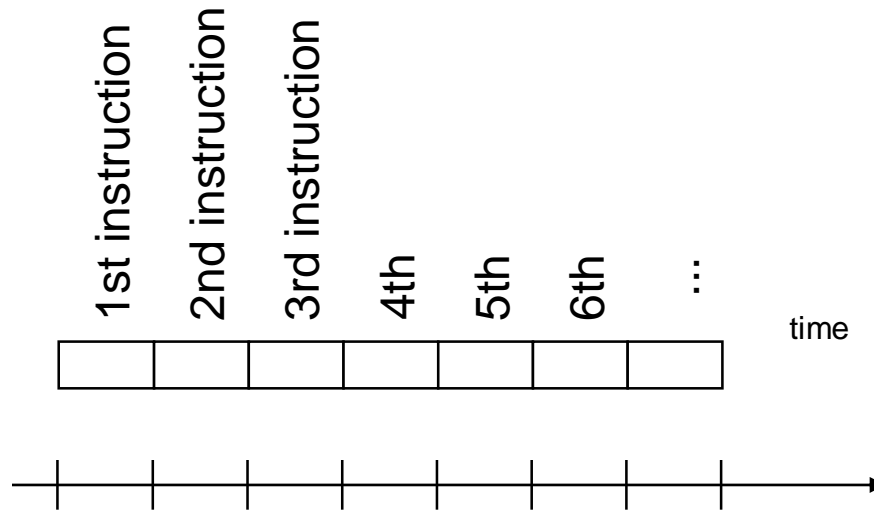
equivalently

$$\begin{array}{l} \text{CPU execution time} \\ \text{for a program} \end{array} = \begin{array}{l} \text{CPU clock cycles} \\ \text{for a program} \end{array} \times \text{Clock cycle time}$$

- So, to improve performance one can either:
  - reduce the number of cycles for a program, or
  - reduce the clock cycle time, or, equivalently,
  - increase the clock rate

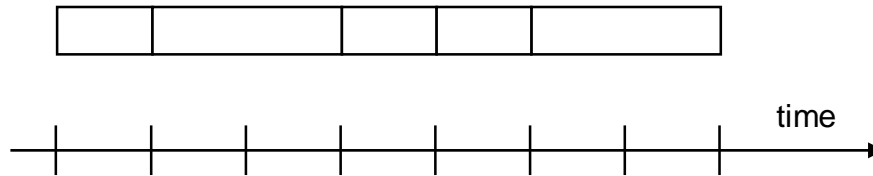
# How many cycles are required for a program?

- Could assume that # of cycles = # of instructions



- *This assumption is incorrect!* Because:
  - Different instructions take different amounts of time (cycles)
  - Why...?

# How many cycles are required for a program?



- 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 because it means changing the hardware design. More later...



# Example

---

- Our favorite program runs in 10 seconds on computer A, which has a 400Mhz. 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?*



# Terminology

---

- A given program will require:
  - some number of instructions (machine instructions)
  - some number of cycles
  - some number of seconds
- We have a vocabulary that relates these quantities:
  - *cycle time* (seconds per cycle)
  - *clock rate* (cycles per second)
  - (*average*) *CPI* (cycles per instruction)
    - a floating point intensive application might have a higher average CPI
  - *MIPS* (millions of instructions per second)
    - this would be higher for a program using simple instructions



# Performance Measure

---

- *Performance is determined by execution time*
- Do any of these 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





# Performance Equation II

---

$$\begin{array}{l} \text{CPU execution time} \\ \text{for a program} \end{array} = \begin{array}{l} \text{Instruction count} \\ \text{for a program} \end{array} \times \text{average CPI} \times \text{Clock cycle time}$$

- *Derive the above equation from Performance Equation I*



# CPI Example I

---

- 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
- *Which 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?*



# CPI Example II

---

- 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 1, 2 and 3 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 500 MHz machine with three different classes of instructions: Class A, Class B, and Class C, which require 1, 2 and 3 cycles (respectively). Both compilers are used to produce code for a large piece of software.
- Compiler 1 generates code with 5 billion Class A instructions, 1 billion Class B instructions, and 1 billion Class C instructions.
- Compiler 2 generates code with 10 billion Class A instructions, 1 billion Class B instructions, and 1 billion 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!
- Benchmark suites
  - Perfect Club: set of application codes
  - Livermore Loops: 24 loop kernels
  - Linpack: linear algebra package
  - SPEC: mix of code from industry organization



# Summary

---

- Performance is specific to a particular program
  - 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

# COD Ch. 3



## Instructions: Language of the Machine

---



# Instructions: Overview

---

- Language of the machine
- More primitive than higher level languages, e.g., no sophisticated control flow such as *while* or *for* loops
- Very restrictive
  - e.g., MIPS arithmetic instructions
- We'll be working with the MIPS instruction set architecture
  - inspired most architectures developed since the 80's
  - used by NEC, Nintendo, Silicon Graphics, Sony
  - the name is just not related to *millions of instructions per second* !
  - it stands for **m**icrocomputer without **i**nterlocked **p**ipeline **s**tages !
- Design goals: *maximize performance* and *minimize cost* and *reduce design time*





# MIPS Arithmetic

---

- All MIPS arithmetic instructions have 3 operands
- Operand order is fixed (e.g., destination first)

- *Example:*

C code:

$A = B + C$

compiler's job to associate  
variables with registers

MIPS code:

add \$s0, \$s1, \$s2



# MIPS Arithmetic

- Design Principle 1: *simplicity favors regularity.*

*Translation: Regular instructions make for simple hardware!*

- *Simpler hardware reduces design time and manufacturing cost.*

- Of course this complicates some things...

C code:

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

MIPS code  
(arithmetic):

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

Allowing variable number of operands would simplify the assembly code but complicate the hardware.

- Performance penalty: high-level code translates to denser machine code.



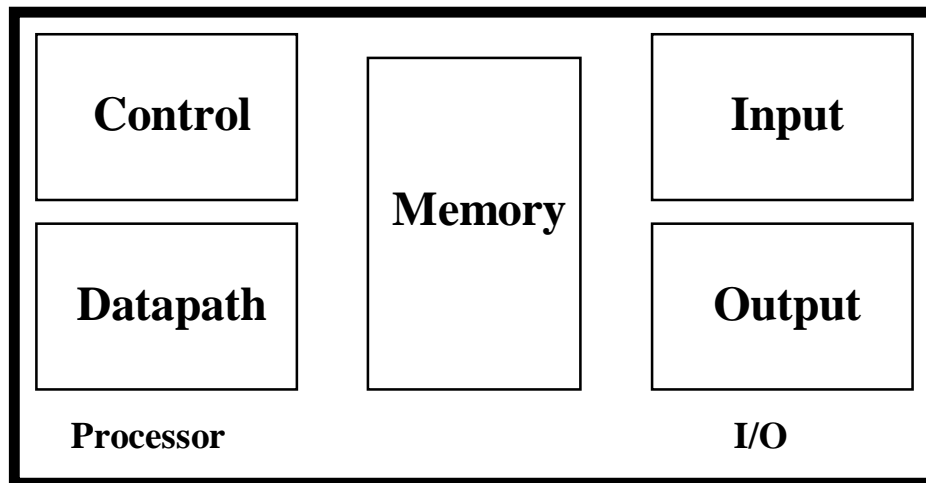
# MIPS Arithmetic

---

- *Operands must be in registers* – only 32 registers provided (which require 5 bits to select one register). Reason for small number of registers:
- Design Principle 2: *smaller is faster*. Why?
  - *Electronic signals have to travel further on a physically larger chip increasing clock cycle time.*
  - *Smaller is also cheaper!*

# Registers vs. Memory

- Arithmetic instructions operands must be in registers
  - MIPS has 32 registers
- Compiler associates variables with registers
- What about programs with lots of variables (arrays, etc.)? Use *memory, load/store* operations to transfer data from memory to register – if not enough registers *spill registers* to memory
- *MIPS is a load/store architecture*





# Memory Organization

---

- Viewed as a large single-dimension array with access by *address*
- A memory address is an *index* into the memory array
- *Byte addressing* means that the index points to a byte of memory, and that the unit of memory accessed by a load/store is a byte

|   |                |
|---|----------------|
| 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 |

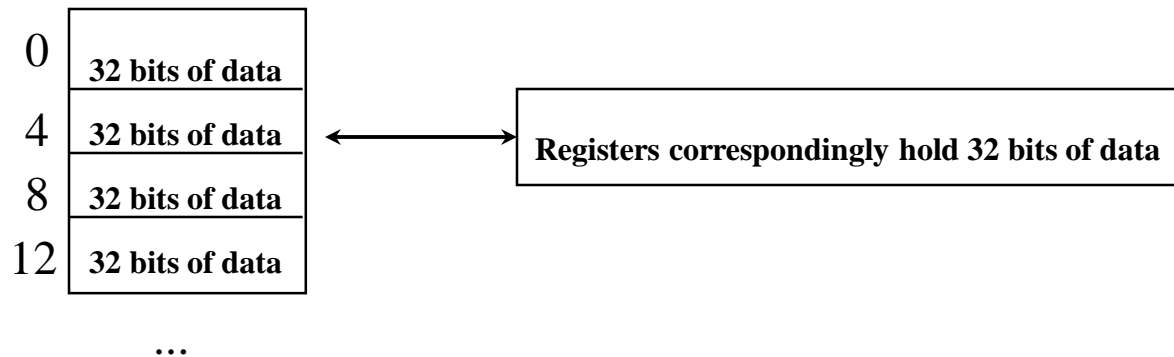
...



# Memory Organization

---

- Bytes are load/store units, but most data items use larger *words*
- For MIPS, a word is 32 bits or 4 bytes.



- $2^{32}$  bytes with byte addresses from 0 to  $2^{32}-1$
- $2^{30}$  words with byte addresses 0, 4, 8, ...  $2^{32}-4$ 
  - i.e., words are *aligned*
  - *what are the least 2 significant bits of a word address?*



# Load/Store Instructions

---

- *Load* and *store* instructions
- *Example:*

C code:                     $A[8] = h + A[8];$

                                 value            offset            address

                                 ↘                    ↘                    ↙

MIPS code    (load):       lw    \$t0, 32(\$s3)

                  (arithmetic):    add   \$t0, \$s2, \$t0

                  (store):        sw    \$t0, 32(\$s3)

- Load word has destination first, store has destination last
- Remember MIPS arithmetic operands are registers, not memory locations
  - therefore, words must first be moved from memory to registers using loads before they can be operated on; then result can be stored back to memory



# So far we've learned:

---

- MIPS

- loading words but addressing bytes
- arithmetic on registers only

- Instruction

## Meaning

add \$s1, \$s2, \$s3

$\$s1 = \$s2 + \$s3$

sub \$s1, \$s2, \$s3

$\$s1 = \$s2 - \$s3$

lw \$s1, 100(\$s2)

$\$s1 = \text{Memory}[\$s2+100]$

sw \$s1, 100(\$s2)

$\text{Memory}[\$s2+100] = \$s1$





# Machine Language

---

- Instructions, like registers and words of data, are also 32 bits long
  - *Example:* `add $t0, $s1, $s2`
  - registers are numbered, e.g., `$t0` is 8, `$s1` is 17, `$s2` is 18
- Instruction Format **R-type** ("R" for aRithmetic):

|                       |  |   |   |                 |   |
|-----------------------|--|---|---|-----------------|---|
| 000000                | 10001                                  | 10010                                   | 01000                                   | 00000           | 100000  |
| op                    | rs                                     | rt                                      | rd                                      | shamt           | funct   |
| opcode -<br>operation | first<br>register<br>source<br>operand | second<br>register<br>source<br>operand | register<br>destin-<br>ation<br>operand | shift<br>amount | function field -<br>selects variant<br>of operation |
| 6 bits                | 5 bits                                 | 5 bits                                  | 5 bits                                  | 5 bits          | 6 bits  |



# Machine Language

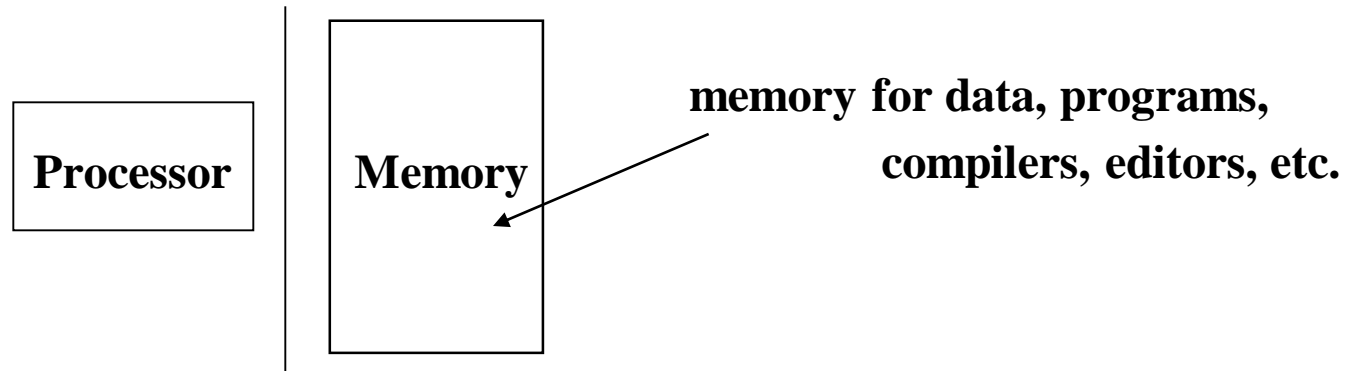
---

- Consider the load-word and store-word instructions,
  - what would the regularity principle have us do?
    - we would have only 5 or 6 bits to determine the offset from a base register - too little...
- Design Principle 3: *Good design demands a compromise*
- Introduce a new type of instruction format
  - **I-type** ("I" for Immediate) for data transfer instructions
  - *Example*: `lw $t0, 1002($s2)`

|        |        |        |                  |
|--------|--------|--------|------------------|
| 100011 | 10010  | 01000  | 0000001111101010 |
| 6 bits | 5 bits | 5 bits | 16 bits          |
| op     | rs     | rt     | 16 bit offset    |

# Stored Program Concept

- *Instructions are bit sequences*, just like data
- 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* (= *execution*)
  - fetch the next instruction and *repeat*



# SPIM – the MIPS simulator

---

- SPIM (MIPS spelt backwards!) is a MIPS simulator that
  - *reads* MIPS assembly language files and *translates* to machine language
  - *executes* the machine language instructions
  - shows contents of *registers* and *memory*
  - works as a *debugger* (supports *break-points* and *single-stepping*)
  - provides basic *OS-like services*, like simple I/O
- SPIM is freely available on-line
- *An important part of our course is to actually write MIPS assembly code and run using SPIM – the only way to learn assembly (or any programming language) is to write lots and lots of code!!!*
- Refer to our material, including slides, on SPIM

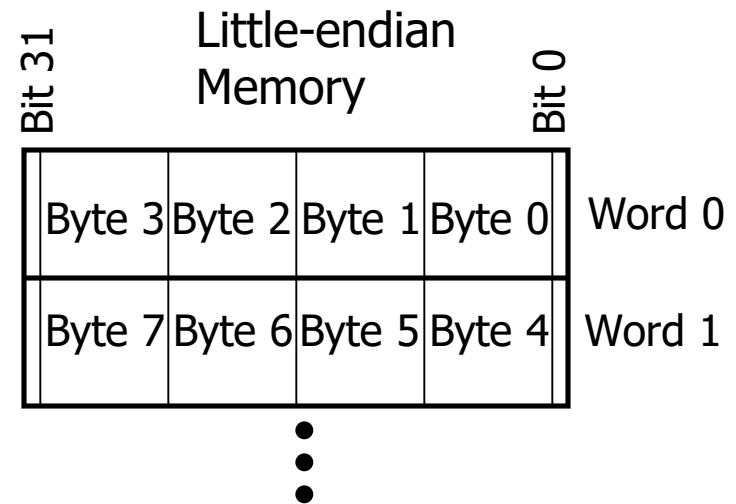
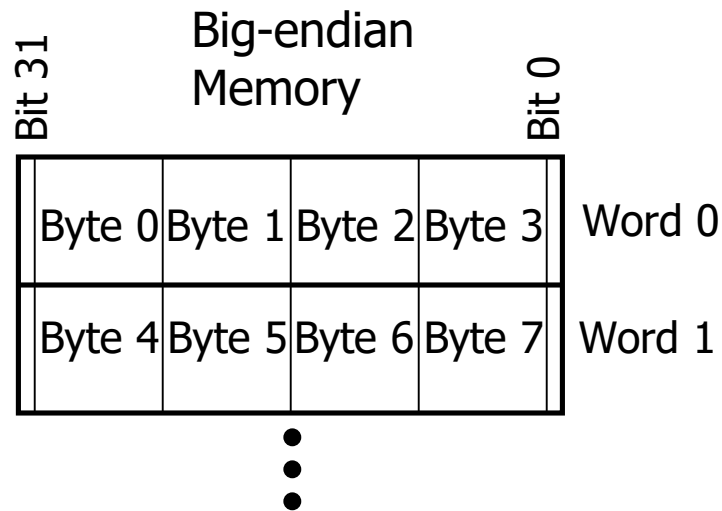
# Memory Organization: Big/Little Endian Byte Order

Bytes in a word can be numbered in two ways:

- byte 0 at the leftmost (most significant) to byte 3 at the rightmost (least significant), called *big-endian*
- byte 3 at the leftmost (most significant) to byte 0 at the rightmost (least significant), called *little-endian*

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 3 | 2 | 1 | 0 |
|---|---|---|---|





# Memory Organization: Big/Little Endian Byte Order

---

- SPIM's memory storage depends on that of the underlying machine
  - *Intel 80x86 processors are **little-endian***
  - because SPIM *always shows* words from left to right a "mental adjustment" has to be made for little-endian memory as in Intel PCs in our labs: start at right of first word go left, start at right of next word go left, ...!
- *Word placement* in memory (from `.data` area of code) or *word access* (`lw`, `sw`) is the same in big or little endian
- *Byte placement* and *byte access* (`lb`, `lbu`, `sb`) depend on big or little endian because of the different numbering of bytes within a word
- *Character placement* in memory (from `.data` area of code) depend on big or little endian because it is equivalent to byte placement after ASCII encoding
- **Run `storeWords.asm` from SPIM examples!!**

# Control: Conditional Branch

- 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
```

} I-type instructions

|        |       |       |                   |
|--------|-------|-------|-------------------|
| 000100 | 01000 | 01001 | 00000000000011001 |
|--------|-------|-------|-------------------|

beq \$t0, \$t1, Label  
(= addr.100)

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

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

```
Label:      . . . .
```

*word-relative addressing:*  
25 words = 100 bytes;  
also *PC-relative* (more...)



# Addresses in Branch

- Instructions:

`bne $t4, $t5, Label`

Next instruction is at Label if  $\$t4 \neq \$t5$

`beq $t4, $t5, Label`

Next instruction is at Label if  $\$t4 = \$t5$

- Format:

|   |    |    |    |               |
|---|----|----|----|---------------|
| I | op | rs | rt | 16 bit offset |
|---|----|----|----|---------------|

- 16 bits is too small a reach in a  $2^{32}$  address space
- Solution: specify a register (as for `lw` and `sw`) and add it to offset
  - use PC (= program counter), called *PC-relative* addressing, based on
  - *principle of locality*: most branches are to instructions near current instruction (e.g., loops and *if* statements)





# Addresses in Branch

---

- Further extend reach of branch by observing all MIPS instructions are a word (= 4 bytes), therefore *word-relative* addressing:
- MIPS branch destination address =  $(\underbrace{PC + 4}) + (4 * \text{offset})$

Because hardware typically increments PC early in execute cycle to point to next instruction

- so offset =  $(\text{branch destination address} - PC - 4)/4$
- *but SPIM does* offset =  $(\text{branch destination address} - PC)/4$

- MIPS unconditional branch instructions:

- *Example:*

$$h=i+j;$$
$$h=i-j;$$

```
add $s3, $s4, $s5
```

```
Lab1: sub $s3, $s4, $s5
```

Lab2: ...

- *Example:* j Label # addr. Label = 100

25 words = 100 bytes

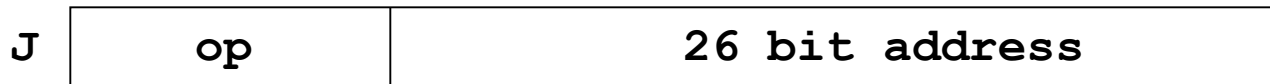
|        |                               |
|--------|-------------------------------|
| 000010 | 00000000000000000000000011001 |
| 6 bits | 26 bits                       |
| op     | 26 bit number                 |



# Addresses in Jump

---

- Word-relative addressing also for jump instructions



- MIPS jump  $j$  instruction replaces *lower* 28 bits of the PC with  $A00$  where  $A$  is the 26 bit address; it *never changes* upper 4 bits
  - *Example:* if  $PC = 1011X$  (where  $X = 28$  bits), it is replaced with  $1011A00$
  - there are  $16(=2^4)$  partitions of the  $2^{32}$  size address space, each partition of size 256 MB ( $=2^{28}$ ), *such that*, in each partition the upper 4 bits of the address is same.
  - if a program crosses an address partition, then a  $j$  that reaches a different partition has to be replaced by  $j_r$  with a full 32-bit address first loaded into the jump register
  - therefore, OS should always try to load a program inside a single partition



# Constants

---

- Small constants are used quite frequently (50% of operands)

e.g.,      $A = A + 5;$   
            $B = B + 1;$   
            $C = C - 18;$

- Solutions? Will these work?
  - create hard-wired registers (like \$zero) for constants like 1
  - put program constants in memory and load them as required

- MIPS Instructions:

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

- *How to make this work?*



# Immediate Operands

---

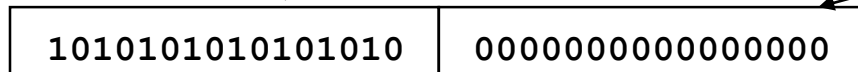
- Make operand part of instruction itself!
- Design Principle 4: *Make the common case fast*
- *Example*: `addi $sp, $sp, 4 # $sp = $sp + 4`

|        |        |        |                   |
|--------|--------|--------|-------------------|
| 001000 | 11101  | 11101  | 00000000000000100 |
| 6 bits | 5 bits | 5 bits | 16 bits           |
| op     | rs     | rt     | 16 bit number     |

# How about larger constants?

- First we need to load a 32 bit constant into a register
- Must use two instructions for this: first new *load upper immediate* instruction for upper 16 bits

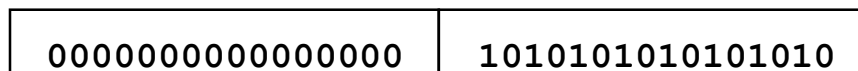
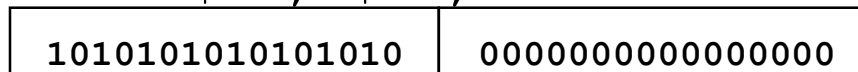
```
lui $t0, 1010101010101010
```



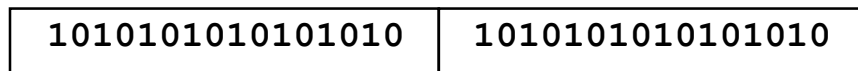
filled with zeros

- Then get lower 16 bits in place:

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



ori



- Now the constant is in place, use register-register arithmetic

# So far

| <u>Instruction</u> | <u>Format</u> | <u>Meaning</u>                         |
|--------------------|---------------|--|
| add \$s1,\$s2,\$s3 | R             | \$s1 = \$s2 + \$s3                     |
| sub \$s1,\$s2,\$s3 | R             | \$s1 = \$s2 - \$s3                     |
| lw \$s1,100(\$s2)  | I             | \$s1 = Memory[\$s2+100]                |
| sw \$s1,100(\$s2)  | I             | Memory[\$s2+100] = \$s1                |
| bne \$s4,\$s5,Lab1 | I             | Next instr. is at Lab1 if \$s4 != \$s5 |
| beq \$s4,\$s5,Lab2 | I             | Next instr. is at Lab2 if \$s4 = \$s5  |
| j Lab3             | J             | Next instr. is at Lab3                 |

## Formats:

|   |         |    |        |        |       |       |
|---|---------|----|--------|--------|-------|-------|
| R | op      | rs | rt     | rd     | shamt | funct |
| I | op      | rs | rt     | 16 bit |       |       |
| J | address | 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
  - *how?* We generate more than one instruction – *pseudo-instruction*
  - can now build general control structures
- The assembler needs a register to manufacture instructions from pseudo-instructions
- There is a *convention* (not mandatory) for use of registers



# Policy-of-Use Convention for Registers



| 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                               |

Register 1, called \$at, is reserved for the assembler; registers 26-27, called \$k0 and \$k1 are reserved for the operating system.



# Assembly Language vs. Machine Language

---

- Assembly provides convenient *symbolic representation*
  - much easier than writing down numbers
  - regular rules: e.g., destination first
- Machine language is the *underlying reality*
  - e.g., destination is no longer first
- Assembly can provide *pseudo-instructions*
  - e.g., `move $t0, $t1` exists only in assembly
  - would be implemented using `add $t0, $t1, $zero`
- When considering performance you should count actual number of machine instructions that will execute



# Procedures

---

- *Example C code:*

```
// procedure adds 10 to input parameter
int main()
{ int i, j;
  i = 5;
  j = add10(i);
  i = j;
  return 0;}

int add10(int i)
{ return (i + 10);}
```

# Procedures

- Translated MIPS assembly
- Note more efficient use of registers possible!

```
.text
```

```
.globl main
```

```
main:
```

```
    addi $s0, $0, 5
```

```
    add  $a0, $s0, $0
```

argument  
to callee

```
    jal add10
```

control returns here

```
    add  $s1, $v0, $0
```

```
    add  $s0, $s1, $0
```

```
    li   $v0, 10
```

```
    syscall
```

} system code  
& call to  
exit

save register  
in stack, see  
figure below

```
add10:
```

```
    addi $sp, $sp, -4  
    sw   $s0, 0($sp)
```

```
    addi $s0, $a0, 10  
    add  $v0, $s0, $0
```

result  
to caller  
restore  
values

```
    lw   $s0, 0($sp)  
    addi $sp, $sp, 4
```

return → jr \$ra

\$sp

MEMORY  
Content of \$s0

High address

Low address

Run this code with PCSpim: procCallsProg1.asm

# MIPS: Software Conventions for Registers

|     |      |                           |
|-----|------|---------------------------|
| 0   | zero | constant 0                |
| 1   | at   | reserved for assembler    |
| 2   | v0   | results from callee       |
| 3   | v1   | returned to caller        |
| 4   | a0   | arguments to callee       |
| 5   | a1   | from caller: caller saves |
| 6   | a2   |                           |
| 7   | a3   |                           |
| 8   | t0   | temporary: caller saves   |
| ... |      | (callee can clobber)      |
| 15  | t7   |                           |

|     |    |                                      |
|-----|----|--------------------------------------|
| 16  | s0 | callee saves                         |
| ... |    | (caller can clobber)                 |
| 23  | s7 |                                      |
| 24  | t8 | temporary (cont'd)                   |
| 25  | t9 |                                      |
| 26  | k0 | reserved for OS kernel               |
| 27  | k1 |                                      |
| 28  | gp | pointer to global area               |
| 29  | sp | stack pointer                        |
| 30  | fp | frame pointer                        |
| 31  | ra | return Address (HW):<br>caller saves |



# Procedures (recursive)

---

- *Example C code* – recursive factorial subroutine:

```
int main()  
{ int i;  
  i = 4;  
  j = fact(i);  
  return 0;}
```

```
int fact(int n)  
{ if (n < 1) return (1);  
  else return ( n*fact(n-1) );}
```

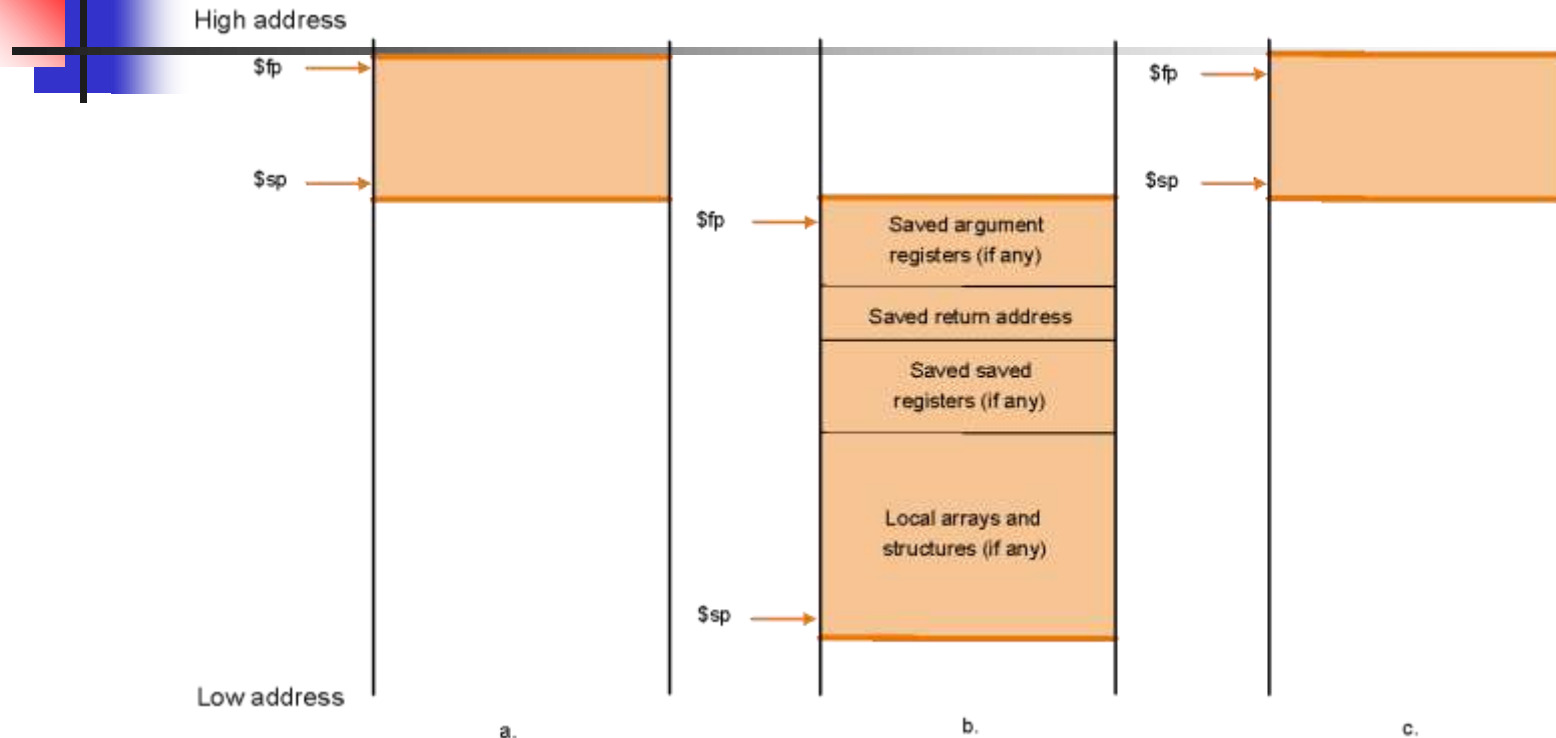
# Procedures (recursive)

## Translated MIPS assembly:

|  |  |   |   |
|--|--|---|---|
|  | <code>.text</code>   |   |   |
|  | <code>.globl main</code>                                       |   |   |
|  | <code>main:</code>   |   |   |
|  | <code>addi \$a0, \$0, 4</code>                                 |   |   |
|  | <code>jal fact</code>  |   |   |
| control<br>returns<br>from fact                    | <code>nop</code>   | branch to<br>L1 if n>=1                                   | <code>slti \$t0, \$a0, 1</code><br><code>beq \$t0, \$0, L1</code><br><code>nop</code>             |
|  | <code>move \$a0, \$v0</code>                                   |   |   |
| print value<br>returned by<br>fact                 | <code>li \$v0, 1</code><br><code>syscall</code>                |   |   |
|  | <code>li \$v0, 10</code><br><code>syscall</code>               |   |   |
| exit   |  |   |   |
|  | <code>fact:</code>   |   |   |
|  | <code>addi \$sp, \$sp, -8</code>                               |   |   |
| save return<br>address and<br>argument in<br>stack | <code>sw \$ra, 4(\$sp)</code><br><code>sw \$a0, 0(\$sp)</code> | if n>=1 call<br>fact recursively<br>with argument<br>n-1  | <code>L1:</code><br><code>addi \$a0, \$a0, -1</code><br><code>jal fact</code><br><code>nop</code> |
|  |  | restore return<br>address, argument,<br>and stack pointer | <code>lw \$a0, 0(\$sp)</code><br><code>lw \$ra, 4(\$sp)</code><br><code>addi \$sp, \$sp, 8</code> |
|  |  | return<br>n*fact(n-1)                                     | <code>mul \$v0, \$a0, \$v0</code>   |
|  |  | return control  | <code>jr \$ra</code>  |

Run this code with PCSpim: factorialRecursive.asm

# Using a Frame Pointer



Variables that are local to a procedure but do not fit into registers (e.g., local arrays, structures, etc.) are also stored in the stack. This area of the stack is the *frame*. The *frame pointer* \$fp points to the top of the frame and the stack pointer to the bottom. The frame pointer does not change during procedure execution, unlike the stack pointer, so it is a stable base register from which to compute offsets to local variables.

Use of the frame pointer is *optional*. If there are no local variables to store in the stack it is not efficient to use a frame pointer.





# Using a Frame Pointer

---

- *Example:* `procCallsProg1Modified.asm`

This program shows code where it may be better to use `$fp`

- Because the stack size is changing, the offset of variables stored in the stack w.r.t. the stack pointer `$sp` changes as well. However, the offset w.r.t. `$fp` would remain constant.
- Why would this be better?

The compiler, when generating assembly, typically maintains a table of program variables and their locations. If these locations are offsets w.r.t `$sp`, then every entry must be updated every time the stack size changes!

- *Exercise:*

*Modify `procCallsProg1Modified.asm` to use a frame pointer*

- Observe that SPIM names register 30 as `s8` rather than `fp`. Of course, you can use it as `fp`, but make sure to initialize it with the same value as `sp`, i.e., `7ffffc`.

# MIPS Addressing Modes

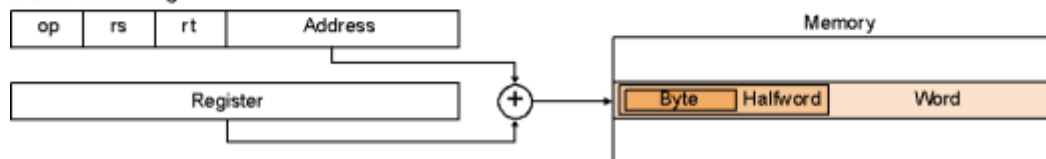
## 1. Immediate addressing



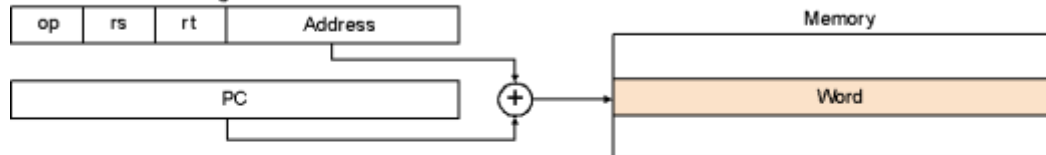
## 2. Register addressing



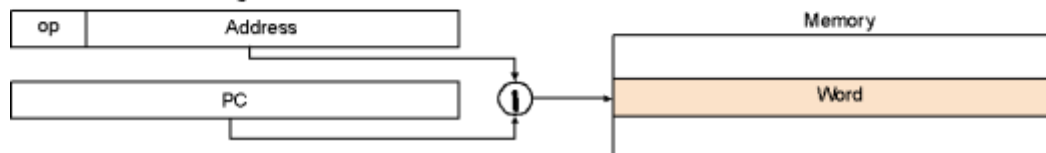
## 3. Base addressing



## 4. PC-relative addressing



## 5. Pseudodirect addressing





# 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

# Summarize MIPS:

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 <sup>30</sup> memory words | Memory[0],<br>Memory[4], ....<br>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 |
| Data transfer      | add immediate           | addi \$s1, \$s2, 100 | $\$s1 = \$s2 + 100$                               | Used to add constants             |
|                    | load word               | lw \$s1, 100(\$s2)   | $\$s1 = \text{Memory}[\$s2 + 100]$                | Word from memory to register      |
|                    | store word              | sw \$s1, 100(\$s2)   | $\text{Memory}[\$s2 + 100] = \$s1$                | Word from register to memory      |
|                    | load byte               | lb \$s1, 100(\$s2)   | $\$s1 = \text{Memory}[\$s2 + 100]$                | Byte from memory to register      |
|                    | store byte              | sb \$s1, 100(\$s2)   | $\text{Memory}[\$s2 + 100] = \$s1$                | 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             | $\$ra = \text{PC} + 4$ ; go to 10000              | For procedure call                |



# 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 *R(educed)ISC* vs. *C(omplex)ISC*
  - virtually all new instruction sets since 1982 have been RISC
- We'll look at PowerPC and 80x86



# PowerPC Special Instructions

---

- Indexed addressing

- *Example:* `lw $t1,$a0+$s3 # $t1=Memory[$a0+$s3]`
- what do we have to do in MIPS?  
`add $t0, $a0, $s3`  
`lw $t1, 0($t0)`

- 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?  
`lw $t0, 4($s3)`  
`addi $s3, $s3, 4`

- Others:

- load multiple words/store multiple words
- a special counter register to improve loop performance:  
`bc Loop, ctrl != 0 # decrement counter, if not 0 goto loop`
- MIPS:  
`addi $t0, $t0, -1`  
`bne $t0, $zero, Loop`



# A dominant architecture: 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”*



# A dominant architecture: 80x86

---

- Complexity
  - instructions from 1 to 17 bytes long
  - one operand *must* act as both a source and destination
  - one operand *may* come from memory
  - several complex addressing modes
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

*“an architecture that is difficult to explain and impossible to love”*

*“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!



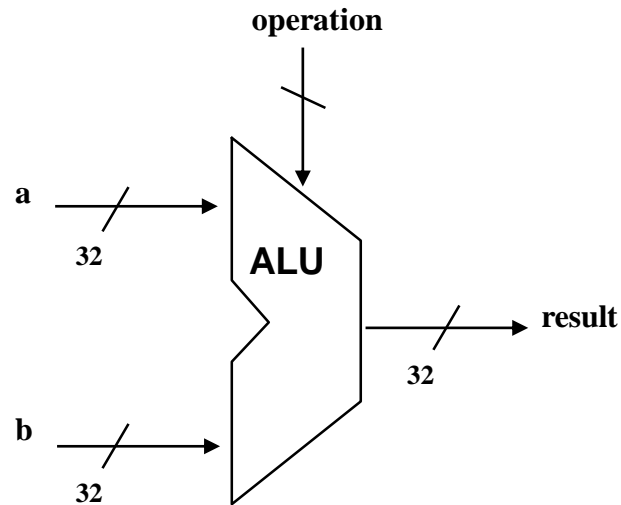
COD Ch. 4

# Arithmetic for Computers

---

# Arithmetic

- Where we've been:
  - performance
  - abstractions
    - *instruction set architecture*
    - *assembly language and machine language*
- What's up ahead:
  - *implementing the architecture*



# MIPS – 2's complement

- 32 bit signed numbers:

|      |      |      |      |      |      |      |      |                 |   |                               |
|------|------|------|------|------|------|------|------|-----------------|---|-------------------------------|
| 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | $_{\text{two}}$ | = | $0_{\text{ten}}$              |
| 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0001 | $_{\text{two}}$ | = | $+1_{\text{ten}}$             |
| 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0010 | $_{\text{two}}$ | = | $+2_{\text{ten}}$             |
| ...  |      |      |      |      |      |      |      |                 |   |                               |
| 0111 | 1111 | 1111 | 1111 | 1111 | 1111 | 1111 | 1110 | $_{\text{two}}$ | = | $+2,147,483,646_{\text{ten}}$ |
| 0111 | 1111 | 1111 | 1111 | 1111 | 1111 | 1111 | 1111 | $_{\text{two}}$ | = | $+2,147,483,647_{\text{ten}}$ |
| 1000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | $_{\text{two}}$ | = | $-2,147,483,648_{\text{ten}}$ |
| 1000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0001 | $_{\text{two}}$ | = | $-2,147,483,647_{\text{ten}}$ |
| 1000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0010 | $_{\text{two}}$ | = | $-2,147,483,646_{\text{ten}}$ |
| ...  |      |      |      |      |      |      |      |                 |   |                               |
| 1111 | 1111 | 1111 | 1111 | 1111 | 1111 | 1111 | 1101 | $_{\text{two}}$ | = | $-3_{\text{ten}}$             |
| 1111 | 1111 | 1111 | 1111 | 1111 | 1111 | 1111 | 1110 | $_{\text{two}}$ | = | $-2_{\text{ten}}$             |
| 1111 | 1111 | 1111 | 1111 | 1111 | 1111 | 1111 | 1111 | $_{\text{two}}$ | = | $-1_{\text{ten}}$             |

maxint

minint

Negative integers are exactly those that have leftmost bit 1

# Two's Complement Operations

- Negation Shortcut: To *negate* any two's complement integer (except for minint) *invert* all bits and *add 1*
  - note that *negate* and *invert* are different operations!
  - *why does this work? Remember we don't know how to add in 2's complement yet! Later...!*
- Sign Extension Shortcut: To convert an n-bit integer into an integer with more than n bits – i.e., to make a narrow integer fill a wider word – *replicate the most significant bit (msb)* of the original number to fill the new bits to its left

- *Example:*    4-bit                      8-bit  
                  0010    =    0000 0010  
                  1010    =    1111 1010

- *why is this correct? Prove!*



# Two's Complement Addition

---

- Perform add just as in junior school (carry/borrow 1s)

- Examples (4-bits):

|             |             |             |             |             |
|-------------|-------------|-------------|-------------|-------------|
| 0101        | 0110        | 1011        | 1001        | 1111        |
| <u>0001</u> | <u>0101</u> | <u>0111</u> | <u>1010</u> | <u>1110</u> |

Do these sums **now!!** Remember all registers are 4-bit including result register!

So you have to **throw away** the carry-out from the msb!!

- Have to beware of *overflow* : if the *fixed* number of bits (4, 8, 16, 32, etc.) in a register *cannot represent the result* of the operation
  - *terminology alert*: overflow *does not mean* there was a carry-out from the msb that we lost (though it sounds like that!) – it means simply that the result in the fixed-sized register is incorrect
    - as can be seen from the above examples there are cases when the result is correct even after losing the carry-out from the msb

# Two's Complement Addition: Verifying Carry/Borrow method

Two (n+1)-bit integers:  $X = x_n X'$ ,  $Y = y_n Y'$

| Carry/borrow<br>add $X + Y$ | $0 \leq X' + Y' < 2^n$<br>(no CarryIn to last bit) | $2^n \leq X' + Y' < 2^{n+1} - 1$<br>(CarryIn to last bit) |
|-----------------------------|--|---|
| $x_n = 0, y_n = 0$          | ok   | not ok (overflow!)  |
| $x_n = 1, y_n = 0$          | ok   | ok  |
| $x_n = 0, y_n = 1$          | ok   | ok  |
| $x_n = 1, y_n = 1$          | not ok (overflow!)                                 | ok  |

- *Prove the cases above!*
- Prove if there is *one more bit* (total n+2 then) available for the result then there is no problem with overflow in add!



# Two's Complement Operations

---

- *Now verify the negation shortcut!*
  - consider  $X + (X + 1) = (X + X) + 1$ :  
associative law – but what if there is overflow in one of the adds on either side, i.e., the result is wrong...!
  - think *minint*!
  - *Examples:*
    - $-0101 = 1010 + 1 = 1011$
    - $-1100 = 0011 + 1 = 0100$
    - $-1000 \neq 0111 + 1 = 1000$





# Detecting Overflow

- *No overflow* when adding a positive and a negative number
- *No overflow* when subtracting numbers with the same sign
- *Overflow occurs* when the result has “wrong” sign (*verify!*):

| Operation | Operand A | Operand B | Result<br>Indicating Overflow |
|-----------|-----------|-----------|-------------------------------|
| $A + B$   | $\geq 0$  | $\geq 0$  | $< 0$                         |
| $A + B$   | $< 0$     | $< 0$     | $\geq 0$                      |
| $A - B$   | $\geq 0$  | $< 0$     | $< 0$                         |
| $A - B$   | $< 0$     | $\geq 0$  | $\geq 0$                      |

- Consider the operations  $A + B$ , and  $A - B$ 
  - *can overflow occur if B is 0 ?*
  - *can overflow occur if A is 0 ?*



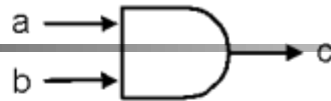
# Effects of Overflow

---

- If an *exception* (interrupt) occurs
  - control jumps to predefined address for exception
  - interrupted address is saved for possible resumption
- Details based on software system/language
  - SPIM: see the EPC and Cause registers
- Don't always want to cause exception on overflow
  - `add, addi, sub` *cause exceptions* on overflow
  - `addu, addiu, subu` *do not cause exceptions* on overflow

# Review: Basic Hardware

1. AND gate ( $c = a \cdot b$ )



| a | b | $c = a \cdot b$ |
|---|---|-----------------|
| 0 | 0 | 0               |
| 0 | 1 | 0               |
| 1 | 0 | 0               |
| 1 | 1 | 1               |

2. OR gate ( $c = a + b$ )



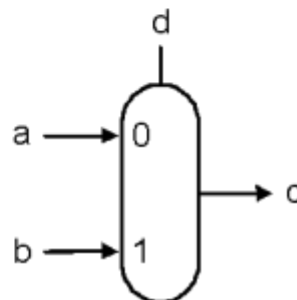
| a | b | $c = a + b$ |
|---|---|-------------|
| 0 | 0 | 0           |
| 0 | 1 | 1           |
| 1 | 0 | 1           |
| 1 | 1 | 1           |

3. Inverter ( $c = \bar{a}$ )



| a | $c = \bar{a}$ |
|---|---------------|
| 0 | 1             |
| 1 | 0             |

4. Multiplexor  
(if  $d = 0$ ,  $c = a$ ;  
else  $c = b$ )



| d | c |
|---|---|
| 0 | a |
| 1 | b |



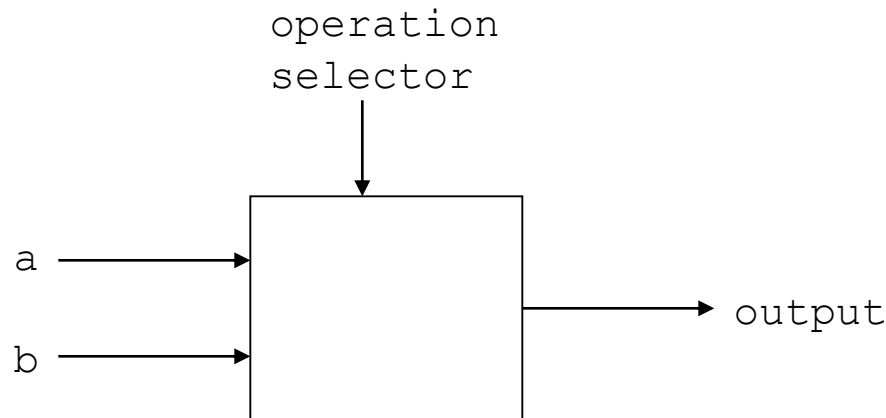
# Review: Boolean Algebra & Gates

---

- *Problem:* Consider logic functions with three inputs: A, B, 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.*

# A Simple Multi-Function Logic Unit

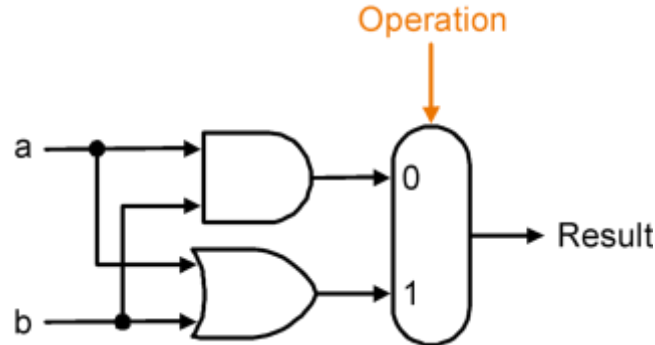
- To warm up let's build a logic unit to support the `and` and `or` instructions for MIPS (32-bit registers)
  - we'll just build a 1-bit unit and use 32 of them



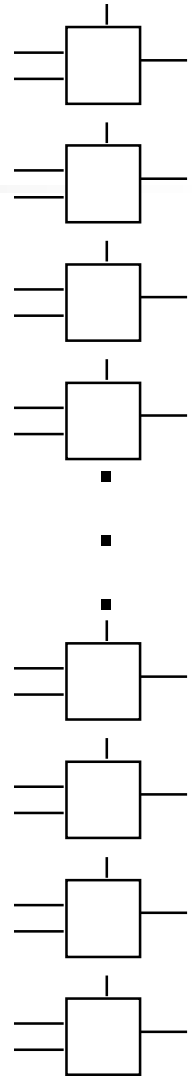
- Possible implementation using a *multiplexor* :

# Implementation with a Multiplexor

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



32 units

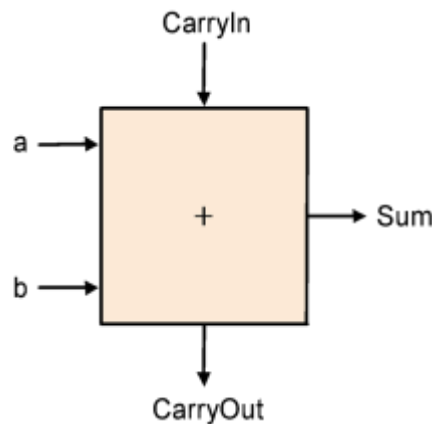


- Lets build our ALU using a MUX (multiplexor):

# Implementations

Not easy to decide the *best* way to implement something

- do not want too many inputs to a single gate
- do not want to have to go through too many gates (= levels)
- for our purposes, ease of comprehension is important
- Let's look at a 1-bit ALU for addition:



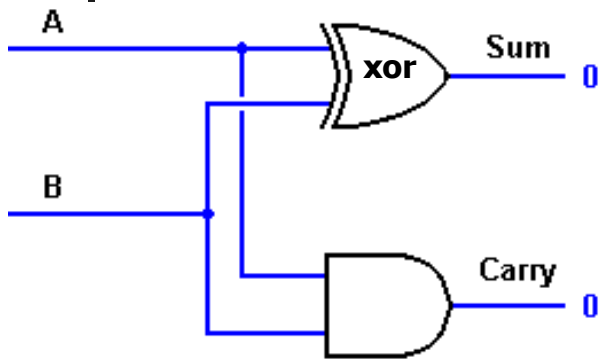
$$C_{out} = a.b + a.c_{in} + b.c_{in}$$

$$\begin{aligned} \text{sum} &= a.\overline{b}.\overline{c_{in}} + \overline{a}.b.\overline{c_{in}} + \\ &\quad \overline{a}.\overline{b}.c_{in} + a.b.c_{in} \\ &= a \oplus b \oplus c_{in} \end{aligned}$$

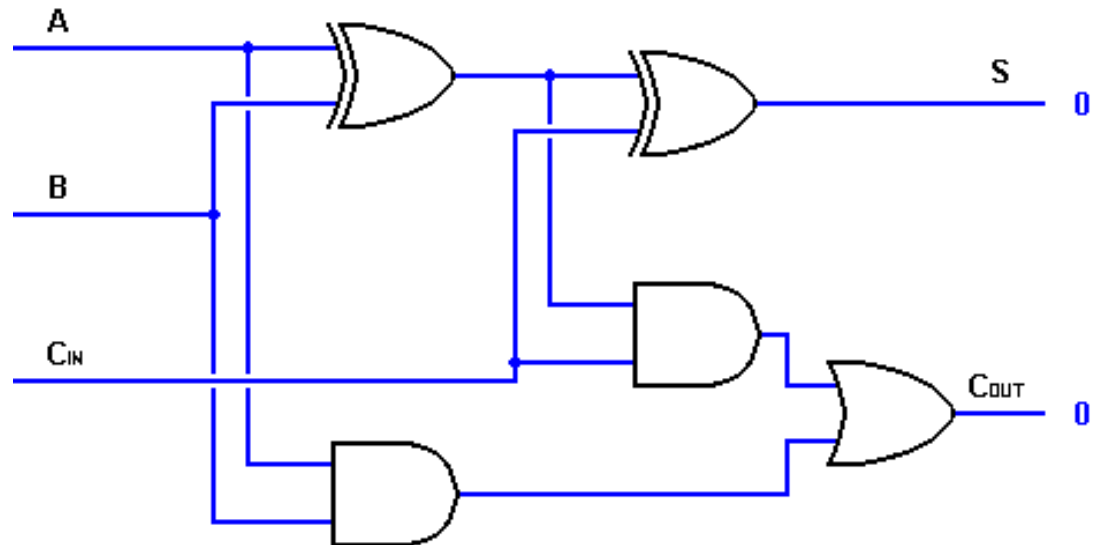
exclusive or (xor)

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

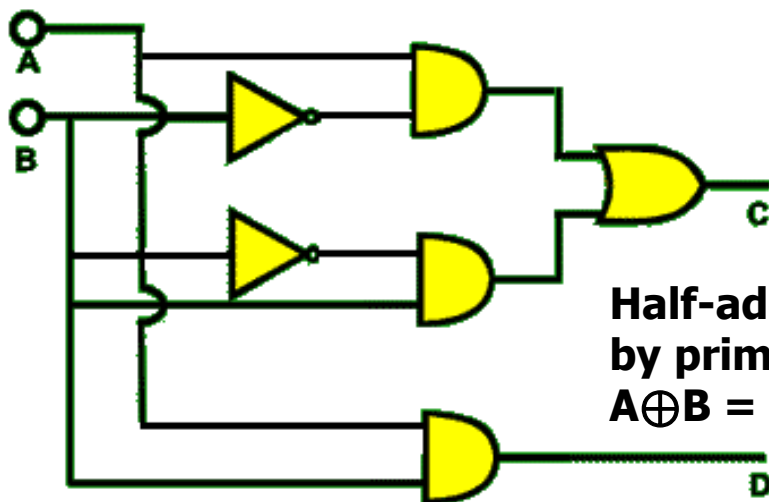
# 1-bit Adder Logic



Half-adder with one xor gate



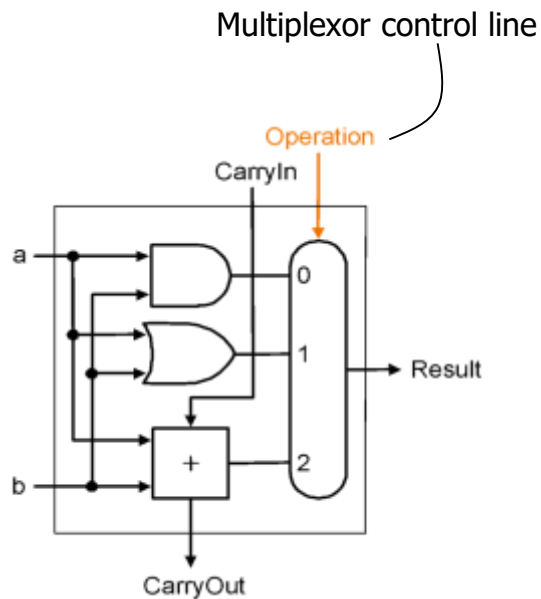
Full-adder from 2 half-adders and an or gate



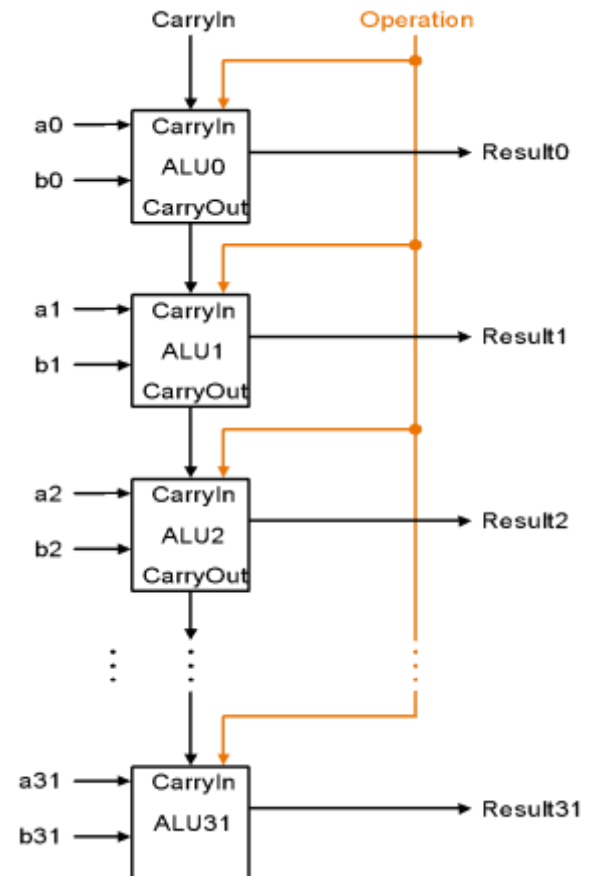
Half-adder with the xor gate replaced by primitive gates using the equation  $A \oplus B = A.B + A.B$



# Building a 32-bit ALU



**1-bit ALU for AND, OR and add**



**Ripple-Carry Logic for 32-bit ALU**



# Multiply

---

- Grade school shift-add method:

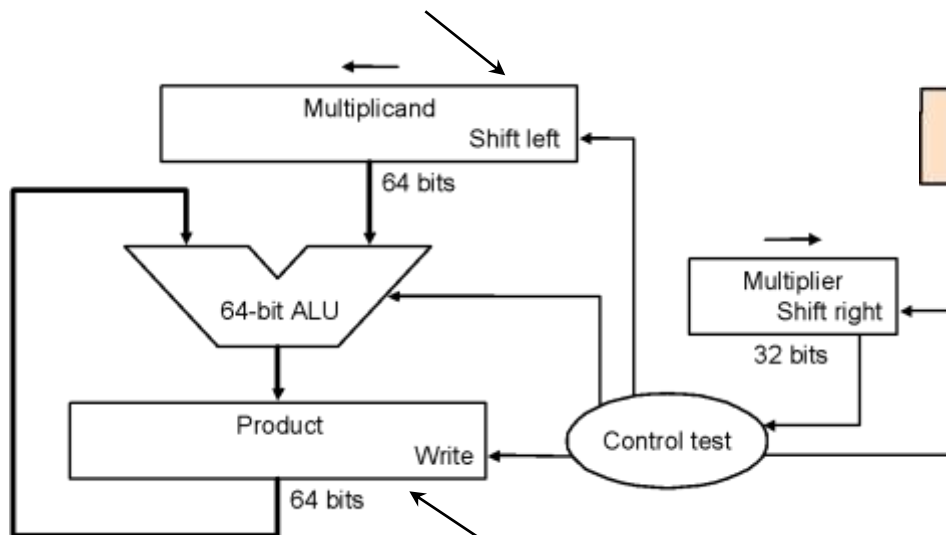
$$\begin{array}{r} \text{Multiplicand} \quad 1000 \\ \text{Multiplier} \quad \times \quad 1001 \\ \hline \quad \quad \quad 1000 \\ \quad \quad 0000 \\ \quad 0000 \\ 1000 \\ \hline \text{Product} \quad 01001000 \end{array}$$

- $m$  bits  $\times$   $n$  bits =  $m+n$  bit product
- Binary makes it easy:
  - multiplier bit 1  $\Rightarrow$  copy multiplicand (1  $\times$  multiplicand)
  - multiplier bit 0  $\Rightarrow$  place 0 (0  $\times$  multiplicand)
- 3 versions of multiply hardware & algorithm:

# Shift-add Multiplier Version

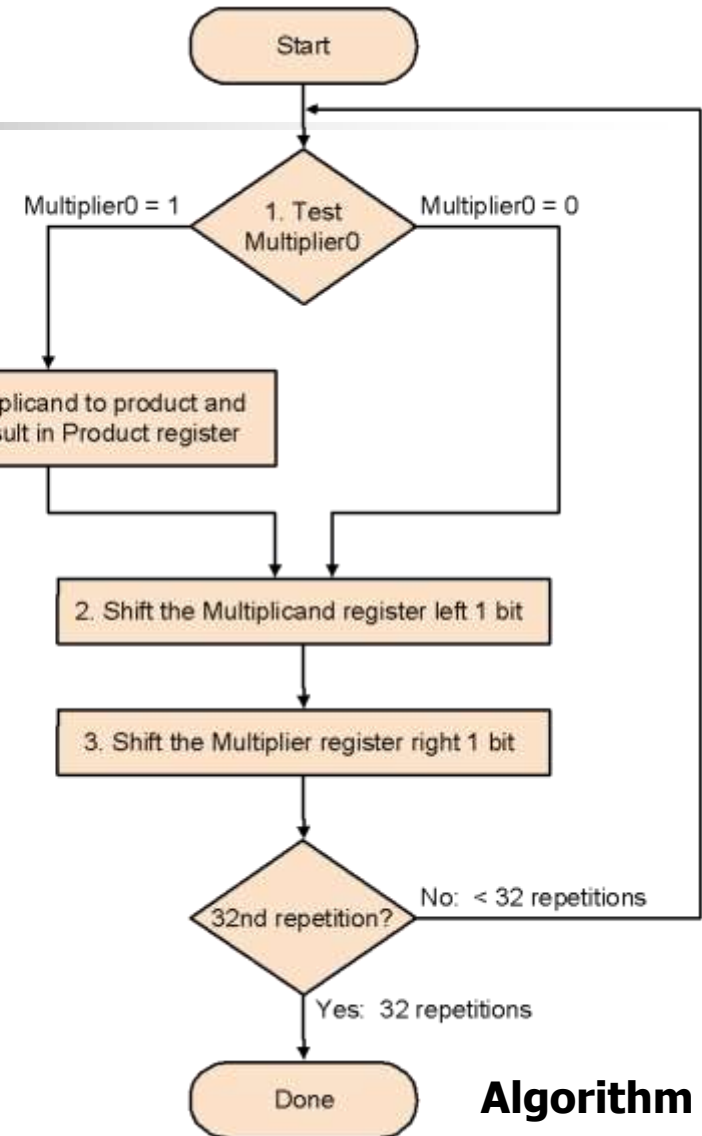
1

32-bit multiplicand starts at right half of multiplicand register



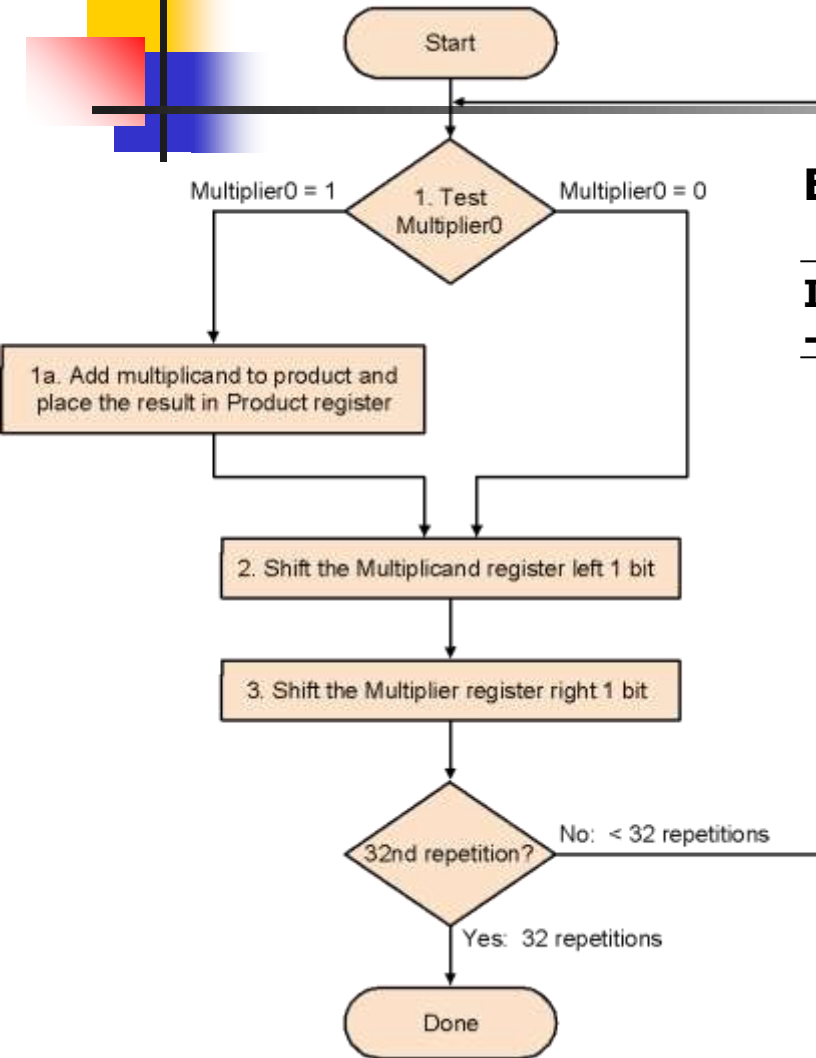
Product register is initialized at 0

**Multiplicand register, product register, ALU are 64-bit wide; multiplier register is 32-bit wide**



**Algorithm**

# Shift-add Multiplier Version1



**Example: 0010 \* 0011:**

| Iteration | Step        | Multiplier | Multiplicand | Product   |
|-----------|-------------|------------|--------------|-----------|
| 0         | init values | 0011       | 0000 0010    | 0000 0000 |
| 1         | 1a          | 0011       | 0000 0010    | 0000 0010 |
|           | 2           | 0011       | 0000 0100    | 0000 0010 |
|           | 3           | 0001       | 0000 0100    | 0000 0010 |
| 2         | ...         |            |              |           |

**Algorithm**

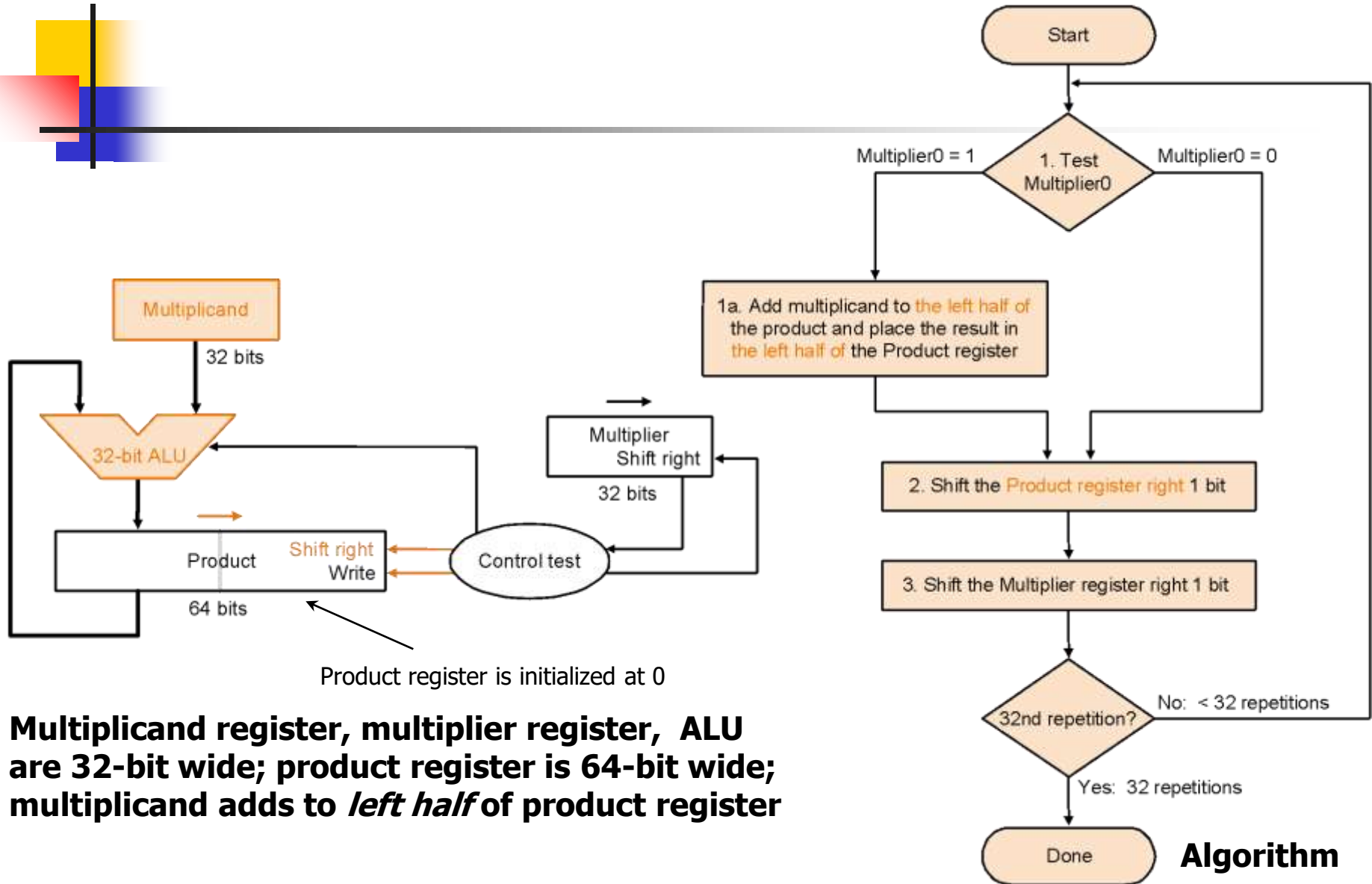
# Observations on Multiply

## Version 1

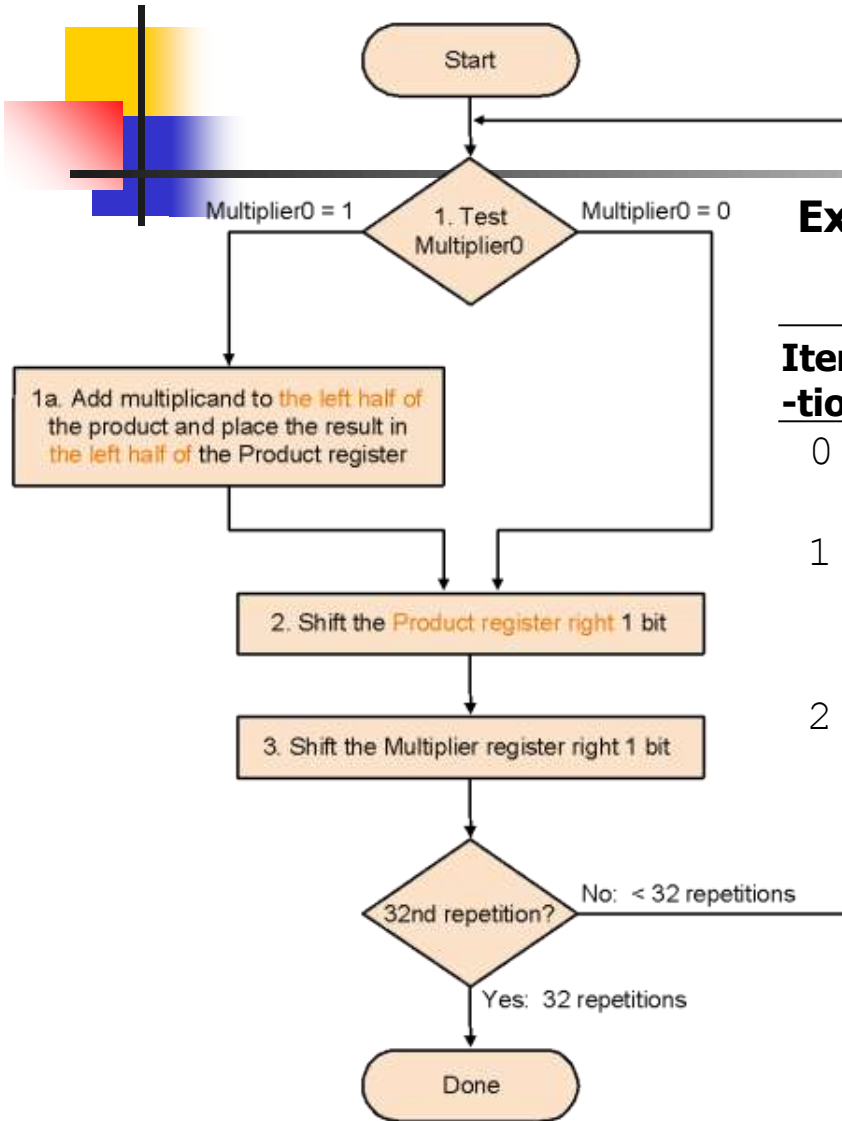
---

- 1 step per clock cycle  $\Rightarrow$  nearly 100 clock cycles to multiply two 32-bit numbers
- Half the bits in the multiplicand register always 0  
 $\Rightarrow$  64-bit adder is wasted
- 0's inserted to right as multiplicand is shifted left  
 $\Rightarrow$  least significant bits of product never change once formed
- Intuition: instead of shifting multiplicand to left, shift product to right...

# Shift-add Multiplier Version 2



# Shift-add Multiplier Version 2



**Example: 0010 \* 0011:**

| Iteration | Step        | Multiplier | Multiplicand | Product   |
|-----------|-------------|------------|--------------|-----------|
| 0         | init values | 0011       | 0010         | 0000 0000 |
| 1         | 1a          | 0011       | 0010         | 0010 0000 |
| 2         | 2           | 0011       | 0010         | 0001 0000 |
| 3         | 3           | 0001       | 0010         | 0001 0000 |
| 2         | ...         |            |              |           |

**Algorithm**

# Observations on Multiply

## Version 2

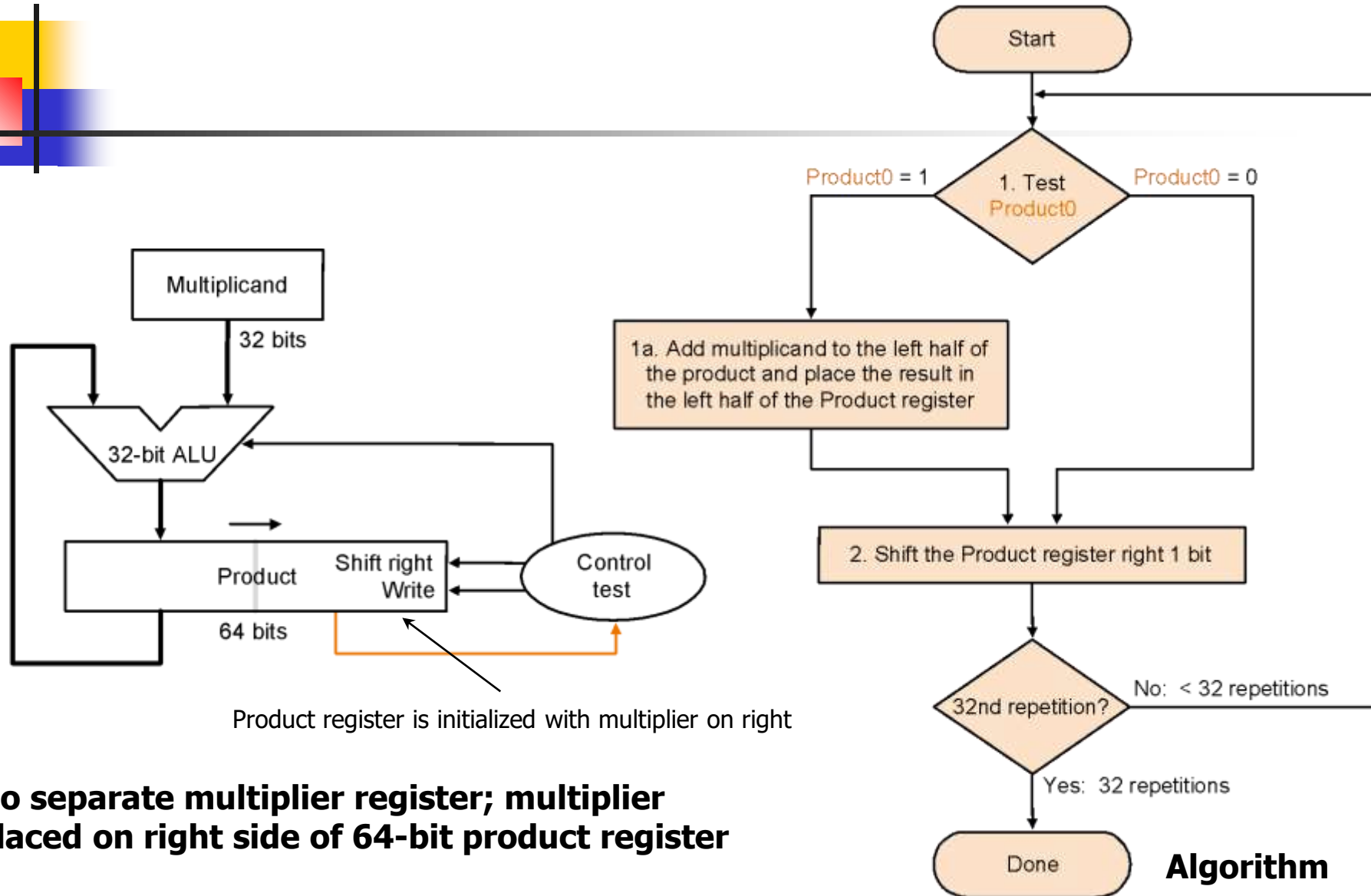


---

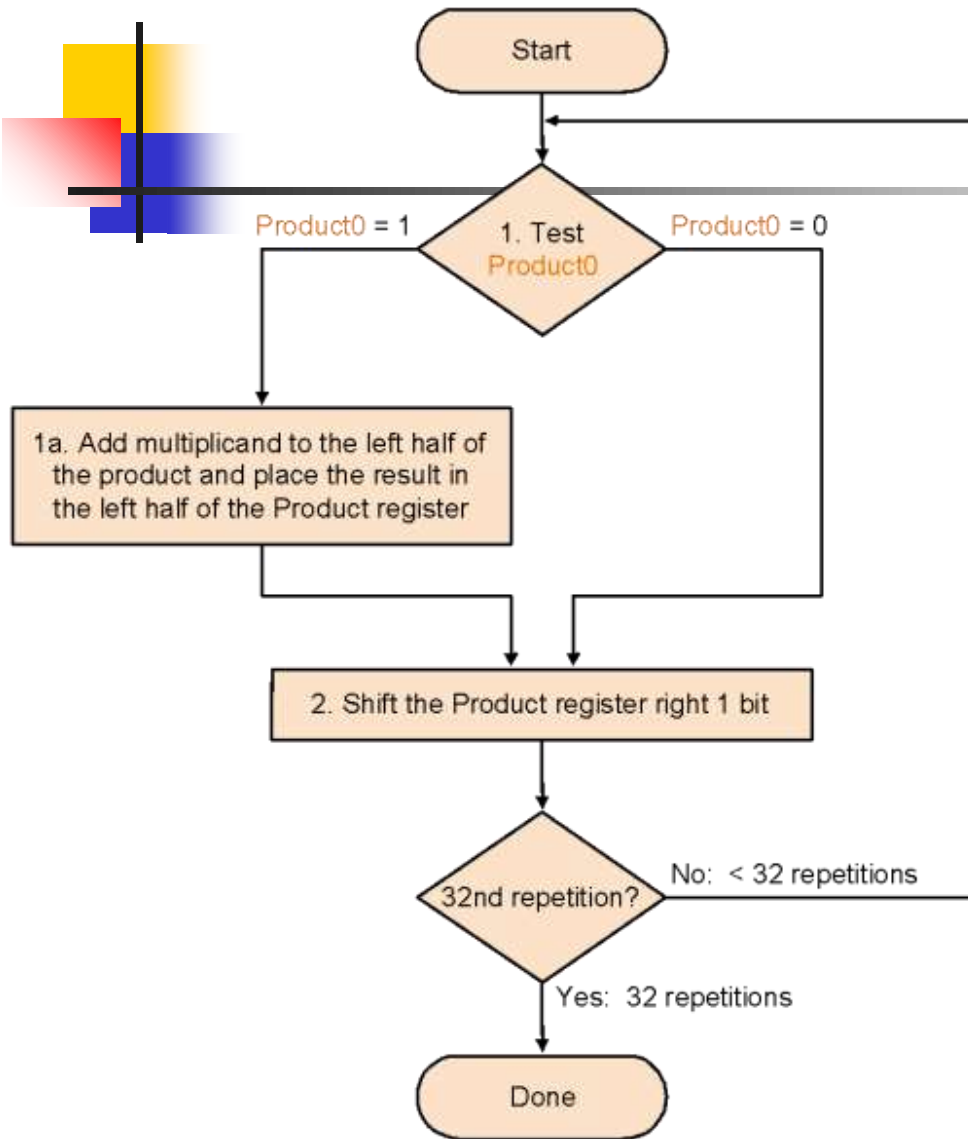
- Each step the product register wastes space that exactly matches the current size of the multiplier
- Intuition: combine multiplier register and product register...



# Shift-add Multiplier Version 3



# Shift-add Multiplier Version 3



**Example: 0010 \* 0011:**

| Iteration | Step        | Multiplicand | Product   |
|-----------|-------------|--------------|-----------|
| 0         | init values | 0010         | 0000 0011 |
| 1         | 1a          | 0010         | 0010 0011 |
| 2         | 2           | 0010         | 0001 0001 |
| 2         | ...         |              |           |

**Algorithm**

# Observations on Multiply

## Version 3



---

- 2 steps per bit because multiplier & product combined
- What about *signed* multiplication?
  - easiest solution is to make both positive and remember whether to negate product when done, i.e., leave out the sign bit, run for 31 steps, then negate if multiplier and multiplicand have opposite signs
- Booth's Algorithm is an elegant way to multiply signed numbers using same hardware – it also often quicker...

# Motivating Booth's algorithm

- Example  $0010 * 0110$ . Traditional:

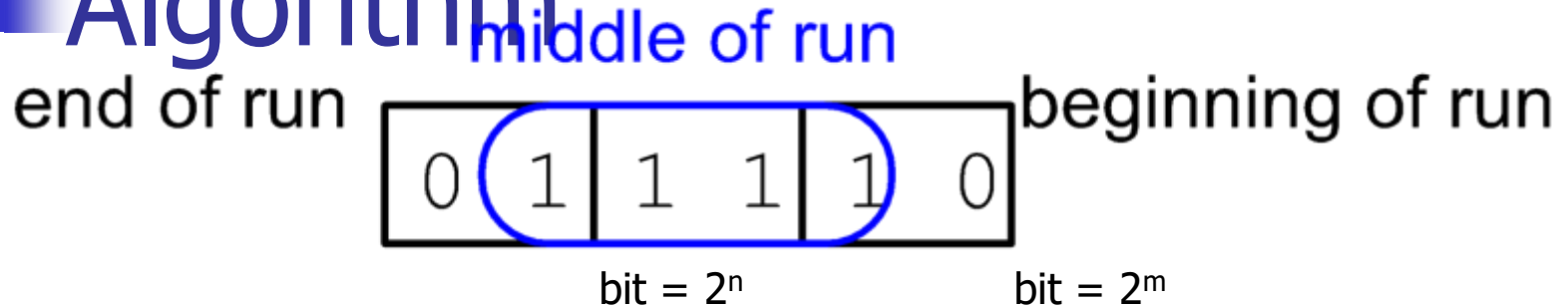
$$\begin{array}{r}
 0010 \\
 \times 0110 \\
 \hline
 0000 \quad \text{shift (0 in multiplier)} \\
 0010 \quad \text{add (1 in multiplier)} \\
 0010 \quad \text{add (1 in multiplier)} \\
 0000 \quad \text{shift (0 in multiplier)} \\
 \hline
 00001100
 \end{array}$$

- Same example. But observe there are two successive 1's in multiplier  $0110 = 2^2 + 2^1 = 2^3 - 2^1$ , so can replace successive 1's by subtract and then add:

$$\begin{array}{r}
 0010 \\
 0110 \\
 \hline
 0000 \quad \text{shift (0 in multiplier)} \\
 -0010 \quad \text{sub (first 1 in multiplier)} \\
 0000 \quad \text{shift (middle of string of 1's)} \\
 0010 \quad \text{add (previous step had last 1)} \\
 \hline
 00001100
 \end{array}$$

# Motivating Booth's

## Algorithm



- Math idea: string of 1's

...0 1 1 ... 1 0... has  
                     successive 1's

value the sum  $2^n + 2^{n-1} + \dots + 2^m = 2^{n+1} - 2^m$

- Replace a string of 1s in multiplier with an *initial subtract when we first see a one* and then later *add after the last one*
  - What if the string of 1's started from the left of the (2's complement) number, e.g., 11110001 – would the formula above have to be modified?!

# Booth from Multiply Version 3

- *Modify Step 1* of the algorithm Multiply Version 3 to consider *2 bits* of the multiplier: *the current bit and the bit to the right* (i.e., the current bit of the previous step). Instead of two outcomes, now there are four:

| <u>Case</u> | <u>Current Bit</u> | <u>Bit to the Right</u> | <u>Explanation</u>  | <u>Example</u>      | <u>Op</u> |
|-------------|--------------------|-------------------------|---------------------|---------------------|-----------|
| 1a          | 0                  | 0                       | Middle of run of 0s | 0 <u>00</u> 1111000 | none      |
| 1b          | 0                  | 1                       | End of run of 1s    | 00 <u>01</u> 111000 | add       |
| 1c          | 1                  | 0                       | Begins run of 1s    | 000111 <u>10</u> 00 | sub       |
| 1d          | 1                  | 1                       | Middle of run of 1s | 00011 <u>11</u> 000 | none      |

- *Modify Step 2* of Multiply Version 3 to *sign extend* when the product is shifted right (*arithmetic right shift*, rather than *logical right shift*) because the product is a signed number
- *Now draw the flowchart for Booth's algorithm !*
- Multiply Version 3 and Booth share the same hardware, *except* Booth requires one extra flipflop to remember the bit to the right of the current bit in the product register – which is the bit pushed out by the preceding right shift

# Booth Example (2 x

| Operation        | Multiplicand | Product     | next?               |
|------------------|--------------|-------------|---------------------|
| 0. initial value | 0010         | 0000 0111 0 | 10 -> sub P = P - M |
| 1c.              | 0010         | 1110 0111 0 | shift P (sign ext)  |
| 2.               | 0010         | 1111 0011 1 | 11 -> nop           |
| 1d.              | 0010         | 1111 0011 1 | shift P (sign ext)  |
| 2.               | 0010         | 1111 1001 1 | 11 -> nop           |
| 1d.              | 0010         | 1111 1001 1 | shift P (sign ext)  |
| 2.               | 0010         | 1111 1100 1 | 01 -> add P = P + M |
| 1b.              | 0010         | 0001 1100 1 | shift P (sign ext)  |
| 2.               | 0010         | 0000 1110 0 | done                |



# Booth Algorithm ( $2 * -3$ )

---

| Operation       | Multiplicand | Product     | next?                 |
|-----------------|--------------|-------------|-----------------------|
| 0.initial value | 0010         | 0000 1101 0 | 10 -> sub $P = P - M$ |
| 1c.             | 0010         | 1110 1101 0 | shift P (sign ext)    |
| 2.              | 0010         | 1111 0110 1 | 01 -> add $P = P + M$ |
| 1b.             | 0010         | 0001 0110 1 | shift P (sign ext)    |
| 2.              | 0010         | 0000 1011 0 | 10 -> sub $P = P - M$ |
| 1c.             | 0010         | 1110 1011 0 | shift P               |
| 2.              | 0010         | 1111 0101 1 | 11 -> nop             |
| 1d.             | 0010         | 1111 0101 1 | shift P               |
| 2.              | 0010         | 1111 1010 1 | done                  |



# Verifying Booth's Algorithm

- multiplier  $a = a_{31} a_{32} \dots a_0$ , multiplicand  $= b$

- $a_i \quad a_{i-1} \quad \text{Operation}$

---

|   |   |       |
|---|---|-------|
| 0 | 0 | nop   |
| 0 | 1 | add b |
| 1 | 0 | sub b |
| 1 | 1 | nop   |

- I.e., if  $a_{i-1} - a_i = \begin{cases} 0, & \text{nop} \\ +1, & \text{add b} \\ -1, & \text{sub b} \end{cases}$

- Therefore, Booth computes sum:

$$\begin{aligned}
 & (a_{-1} - a_0) * b * 2^0 \\
 & + (a_0 - a_1) * b * 2^1 \\
 & + (a_1 - a_2) * b * 2^2 \\
 & \dots \\
 & + (a_{30} - a_{31}) * b * 2^{31} \\
 & = \dots \textit{simplify telescopic sum!} \dots
 \end{aligned}$$



# MIPS Notes

---

- MIPS provides two 32-bit registers  $H_i$  and  $L_o$  to hold a 64-bit product
- `mult`, `multu` (unsigned) put the product of two 32-bit register operands into  $H_i$  and  $L_o$ : overflow is ignored by MIPS but can be detected by programmer by examining contents of  $H_i$
- `mflo`, `mfhi` moves content of  $H_i$  or  $L_o$  to a general-purpose register
- Pseudo-instructions `mul` (without overflow), `mulo` (with overflow), `mulou` (unsigned with overflow) take three 32-bit register operands, putting the product of two registers into the third

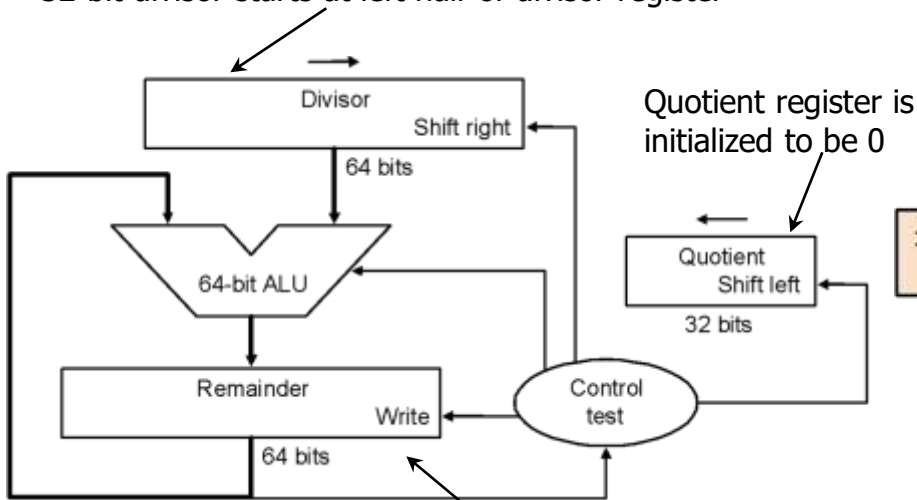
# Divide

|         |      |              |           |
|---------|------|--------------|-----------|
|         |      | 1001         | Quotient  |
| Divisor | 1000 | 1001010      | Dividend  |
|         |      | <u>-1000</u> |           |
|         |      | 10           |           |
|         |      | 101          |           |
|         |      | 1010         |           |
|         |      | <u>-1000</u> |           |
|         |      | 10           | Remainder |

- Junior school method: see how big a multiple of the divisor can be subtracted, creating quotient digit at each step
- Binary makes it easy  $\Rightarrow$  *first*, try  $1 * \text{divisor}$ ; *if too big*,  $0 * \text{divisor}$
- Dividend = (Quotient \* Divisor) + Remainder
- 3 versions of divide hardware & algorithm:

# Divide Version 1

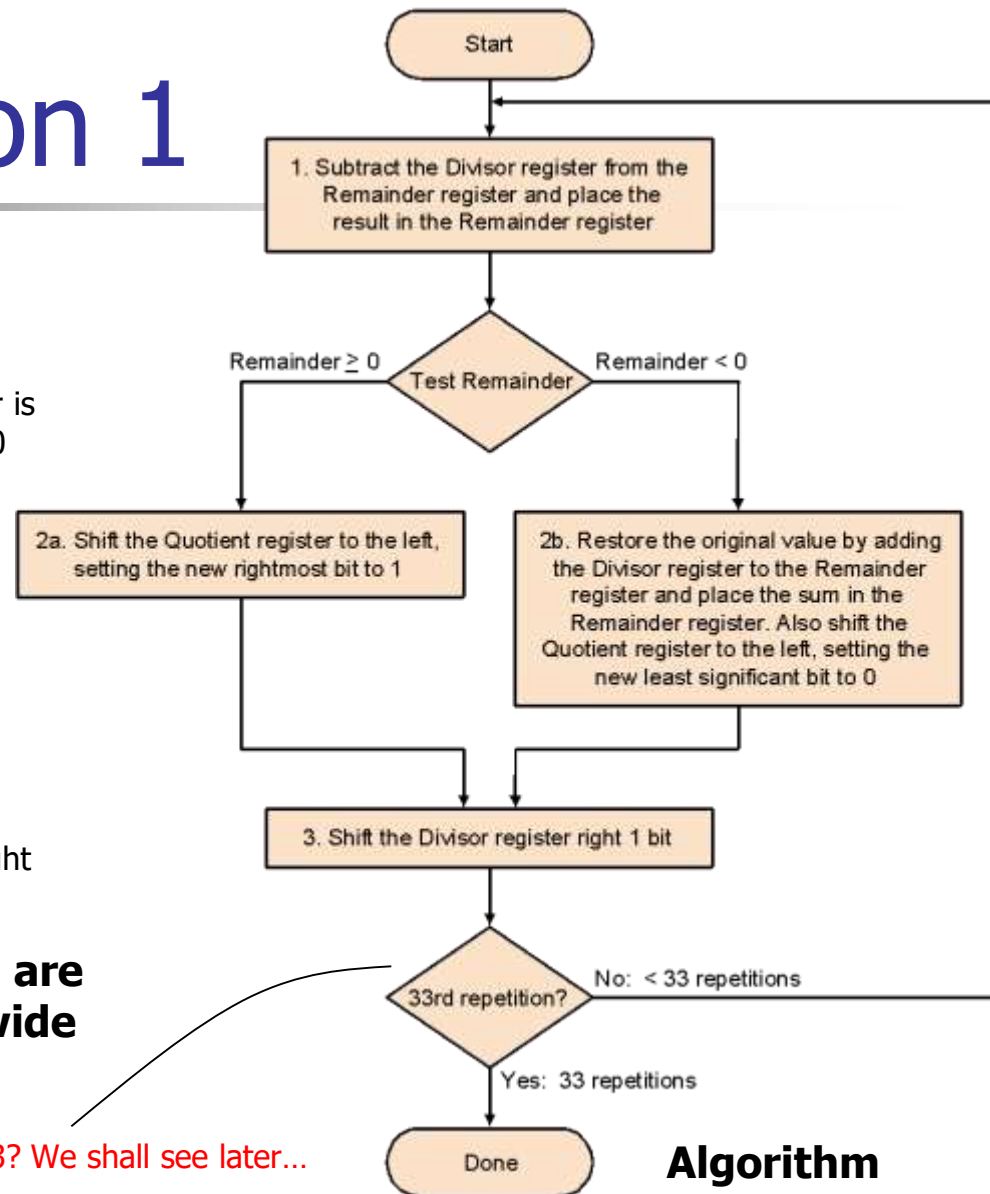
32-bit divisor starts at left half of divisor register



Remainder register is initialized with the dividend at right

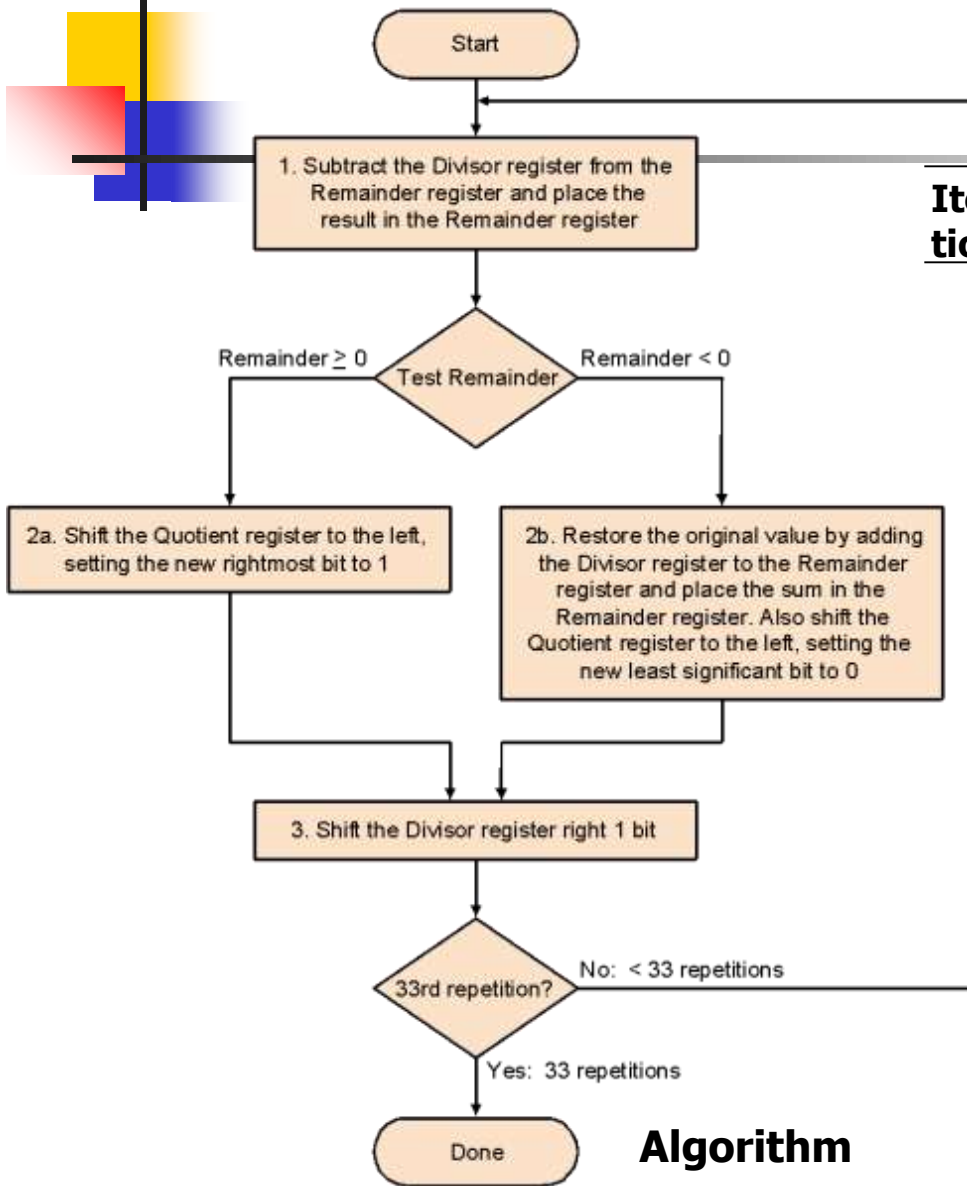
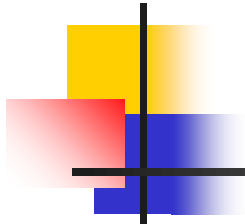
**Divisor register, remainder register, ALU are 64-bit wide; quotient register is 32-bit wide**

Why 33? We shall see later...



**Algorithm**

# Divide Version 1



**Example: 0111 / 0010:**

| Iteration | Step | Quotient | Divisor   | Remainder |
|-----------|------|----------|-----------|-----------|
| 0         | init | 0000     | 0010 0000 | 0000 0111 |
| 1         | 1    | 0000     | 0010 0000 | 1110 0111 |
|           | 2b   | 0000     | 0010 0000 | 0000 0111 |
|           | 3    | 0000     | 0001 0000 | 0000 0111 |
| 2         | ...  |          |           |           |
| 3         |      |          |           |           |
| 4         |      |          |           |           |
| 5         |      |          |           |           |

**Algorithm**

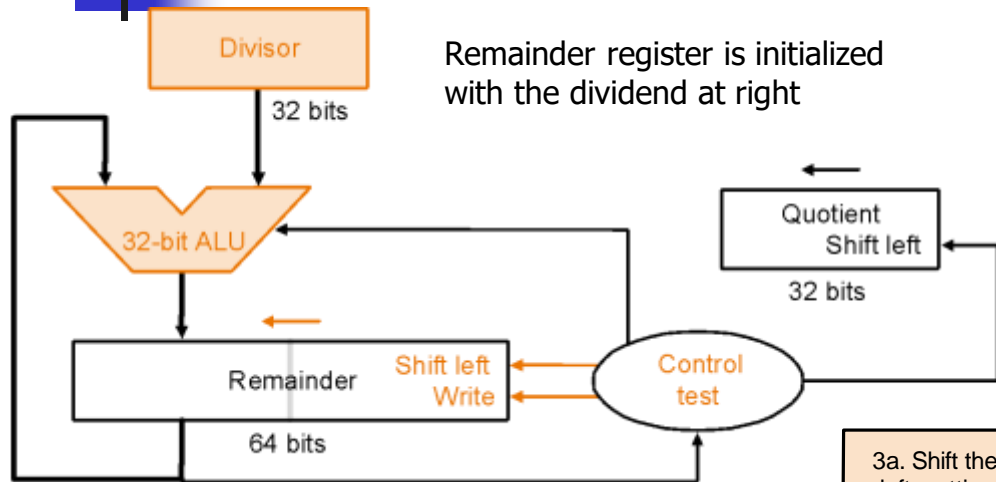


# Observations on Divide Version 1

---

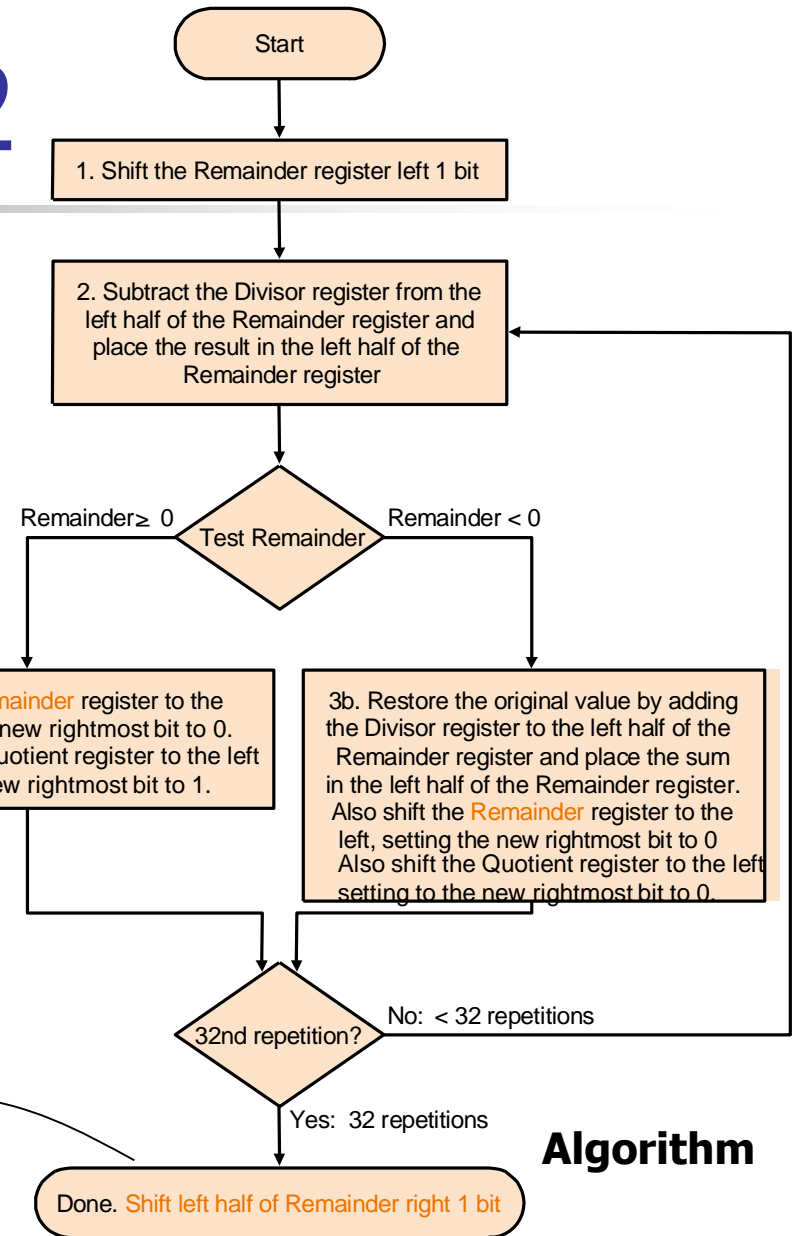
- Half the bits in divisor always 0
  - $\Rightarrow$  1/2 of 64-bit adder is wasted
  - $\Rightarrow$  1/2 of divisor register is wasted
- Intuition: instead of shifting divisor to right, shift remainder to left...
- Step 1 cannot produce a 1 in quotient bit – as all bits corresponding to the divisor in the remainder register are 0 (remember all operands are 32-bit)
- Intuition: switch order to shift first and then subtract – can save 1 iteration...

# Divide Version 2



**Divisor register, quotient register, ALU are 32-bit wide; remainder register is 64-bit wide**

Why this correction step? We shall see later...



**Algorithm**



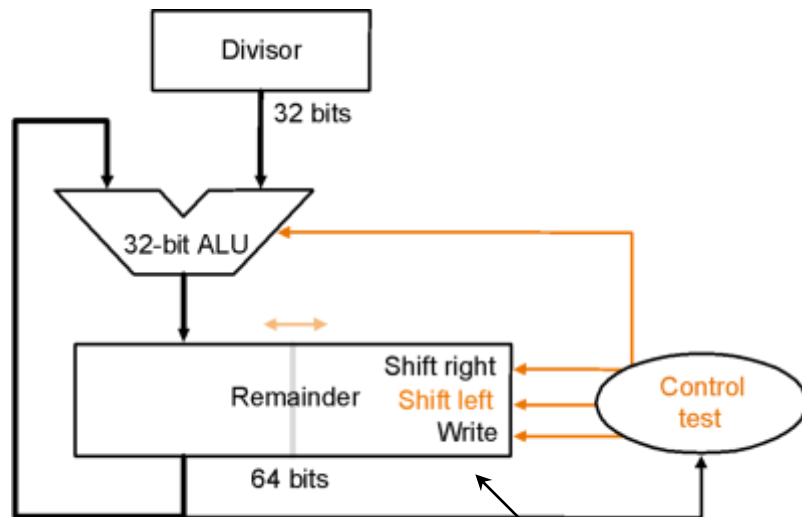
# Observations on Divide Version 2

---

- Each step the remainder register wastes space that exactly matches the current size of the quotient
- Intuition: combine quotient register and remainder register...



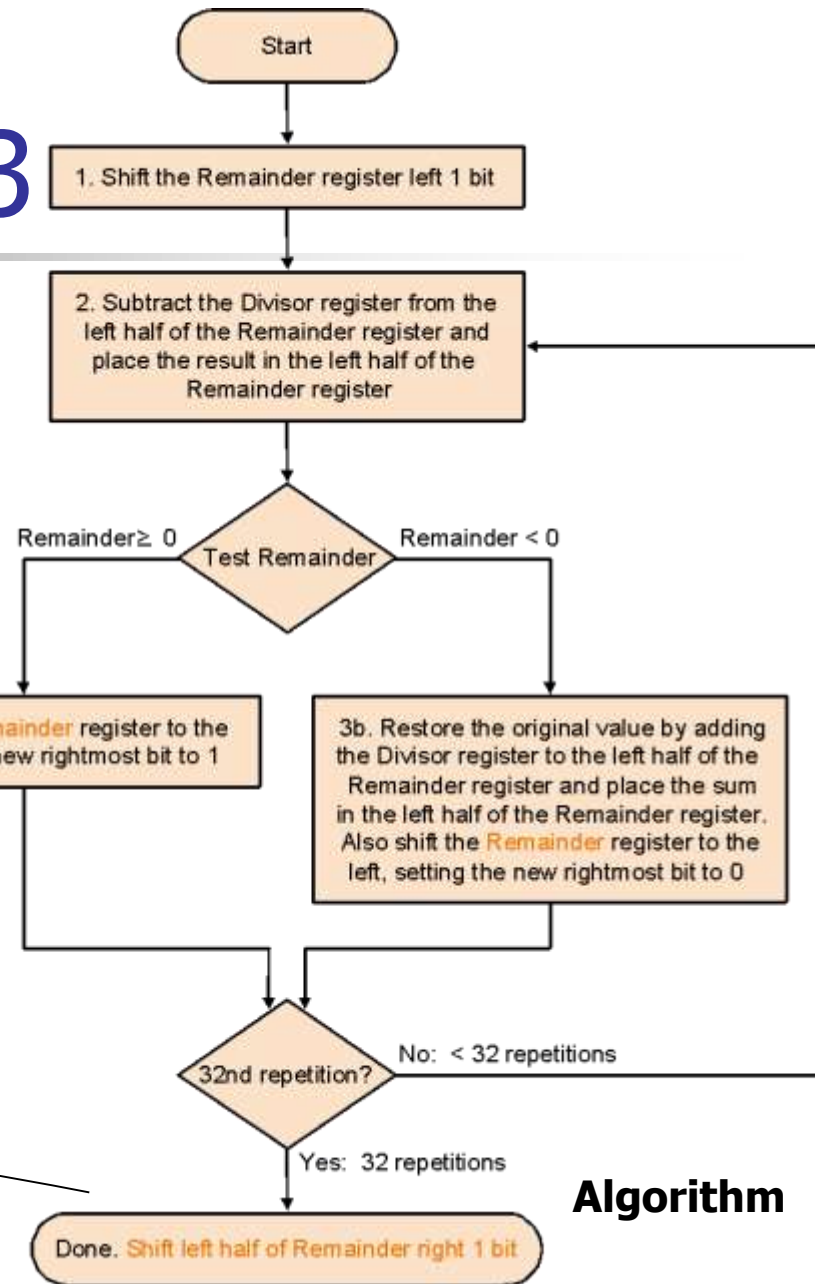
# Divide Version 3



Remainder register is initialized with the dividend at right

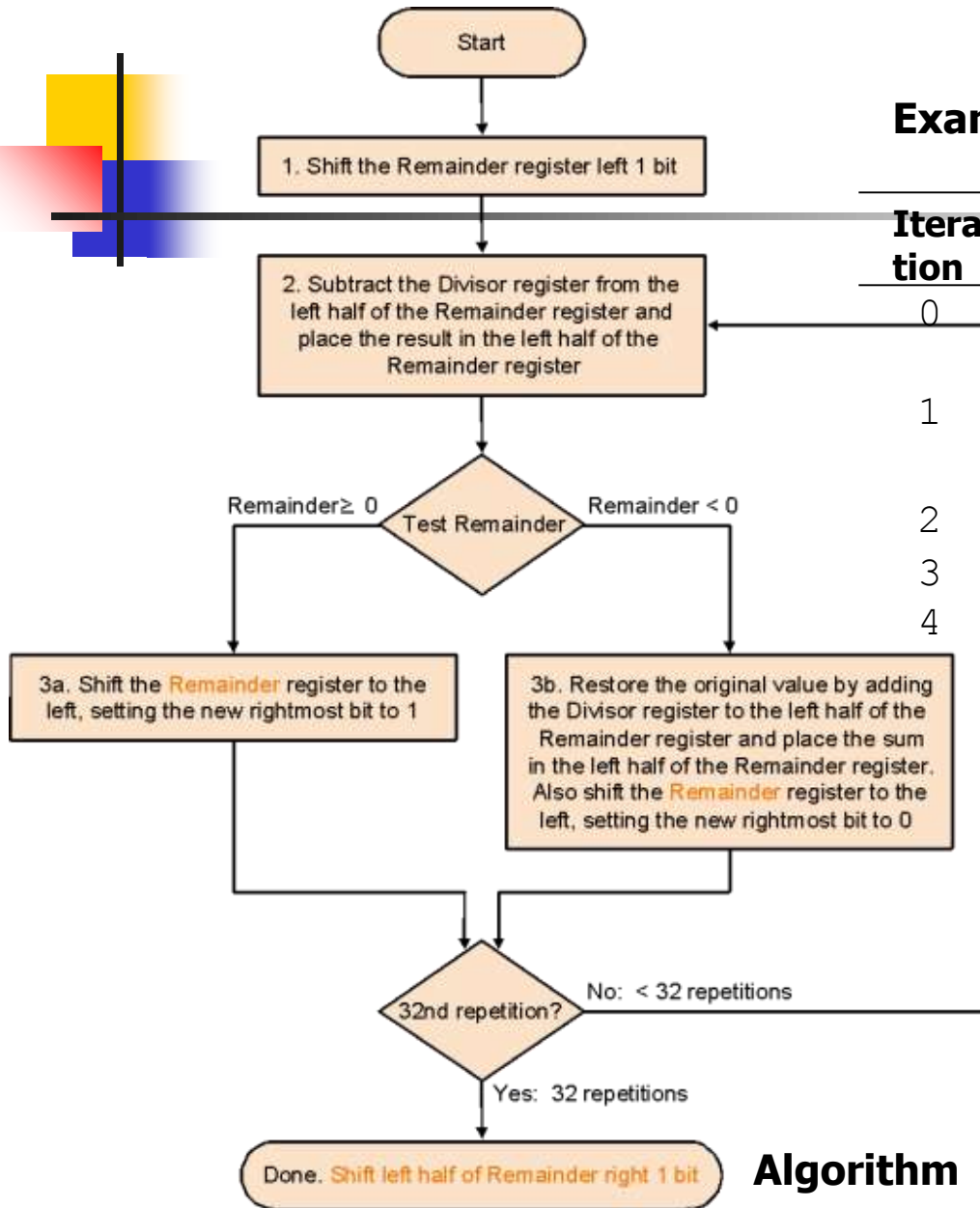
**No separate quotient register; quotient is entered on the right side of the 64-bit remainder register**

Why this correction step? We shall see later...



**Algorithm**

# Divide Version 3



**Example: 0111 / 0010:**

| Iteration | Step | Divisor | Remainder |
|-----------|------|---------|-----------|
| 0         | init | 0010    | 0000 0111 |
| 1         | 1    | 0010    | 0000 1110 |
| 1         | 2    | 0010    | 1110 1110 |
| 1         | 3b   | 0010    | 0001 1100 |
| 2         | ...  |         |           |
| 3         |      |         |           |
| 4         |      |         |           |

**Algorithm**

# Number of Iterations

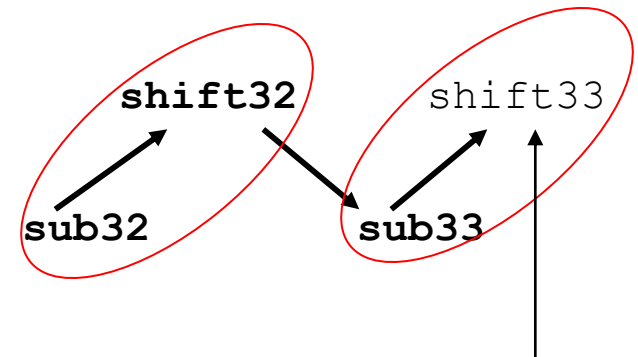
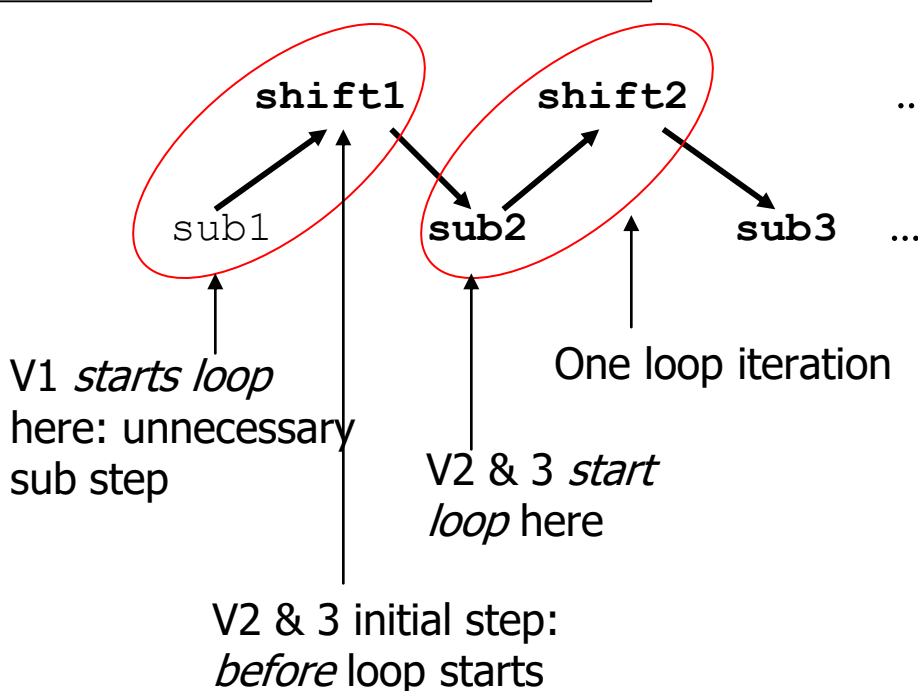
Why the extra iteration in Version 1?

Why the final correction step in Versions 2 & 3?

Ovals represent loop iterations

Shift: see the version descriptions for which registers are shifted

Main insight –  $\text{sub}(i+1)$  must actually follow shift of the divisor (or remainder, depending on version) and the resulting bit in the quotient appears on  $\text{shift}(i+1)$



*Critical situation!* Only the quotient shift is necessary as it corresponds to the outcome of the previous sub. So V1 is ok even though the last divisor shift is redundant, as final divisor is ignored any way; V2 & 3 *must repair remainder* as it has shifted left one time too many



# Observations on Divide Version 3

---

- *Same hardware as Multiply Version 3*
- *Signed divide:*
  - make both divisor and dividend positive and perform division
  - negate the quotient if divisor and dividend were of opposite signs
  - make the sign of the remainder match that of the dividend
  - this ensures always
    - $\text{dividend} = (\text{quotient} * \text{divisor}) + \text{remainder}$
    - $-\text{quotient} (x/y) = \text{quotient} (-x/y)$  (e.g.  $7 = 3*2 + 1$  &  $-7 = -3*2 - 1$ )