COMP 273
Introduction to Computer Systems

COMP 273

CPU & Supporting Chip Sets

Micro Architecture

Part 4

Prof. Joseph Vybihal

Readings

- Soul Of A New Machine

- Following the algorithms from this lecture:
  Divide 15 by 2
  Multiply 12 by 7

- Web Resources:
  - www.mcs.vuw.ac.nz/courses/COMP203/2007T1/Handouts/LectureNotes/lec13.pdf
  - courses.cs.vt.edu/~cs2504/spring2007/lectures/lec13.pdf

COMP 273
Introduction to Computer Systems

Part 1

The CPU with Supporting Hardware

Chip Set

- Supporting chips and circuitry
  - The CPU cannot do everything on its own

A CPU is a powerful machine but it is specialized to do general purpose computations. Specifically ALU and Memory operations. The computer is much more than that and therefore has supporting hardware.

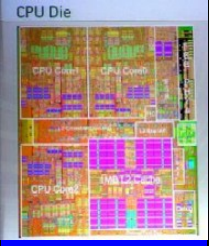
Chip Set

- "On Die" chip sets
  - Circuitry on the CPU die

- "On Board" chip sets
  - Circuitry on the system board
    - Commonly near the CPU

The CPU System

- The CPU (general purpose registers, ALU, cache)
- Supporting OS Registers
- Supporting System-Board Registers
- Supporting CPU Chip-Sets
  - Co-processors
    - Eg: Math, matrix and graphics GPUs
  - ROMS
    - Built-in support for video & basic graphics
    - ASCII support
    - Various communication ports
    - Basic peripheral support



Special assembler instructions

ROM ROM

Like: MMX
ASCII
System BUS

Supporting registers
+ GP registers

Co-processor

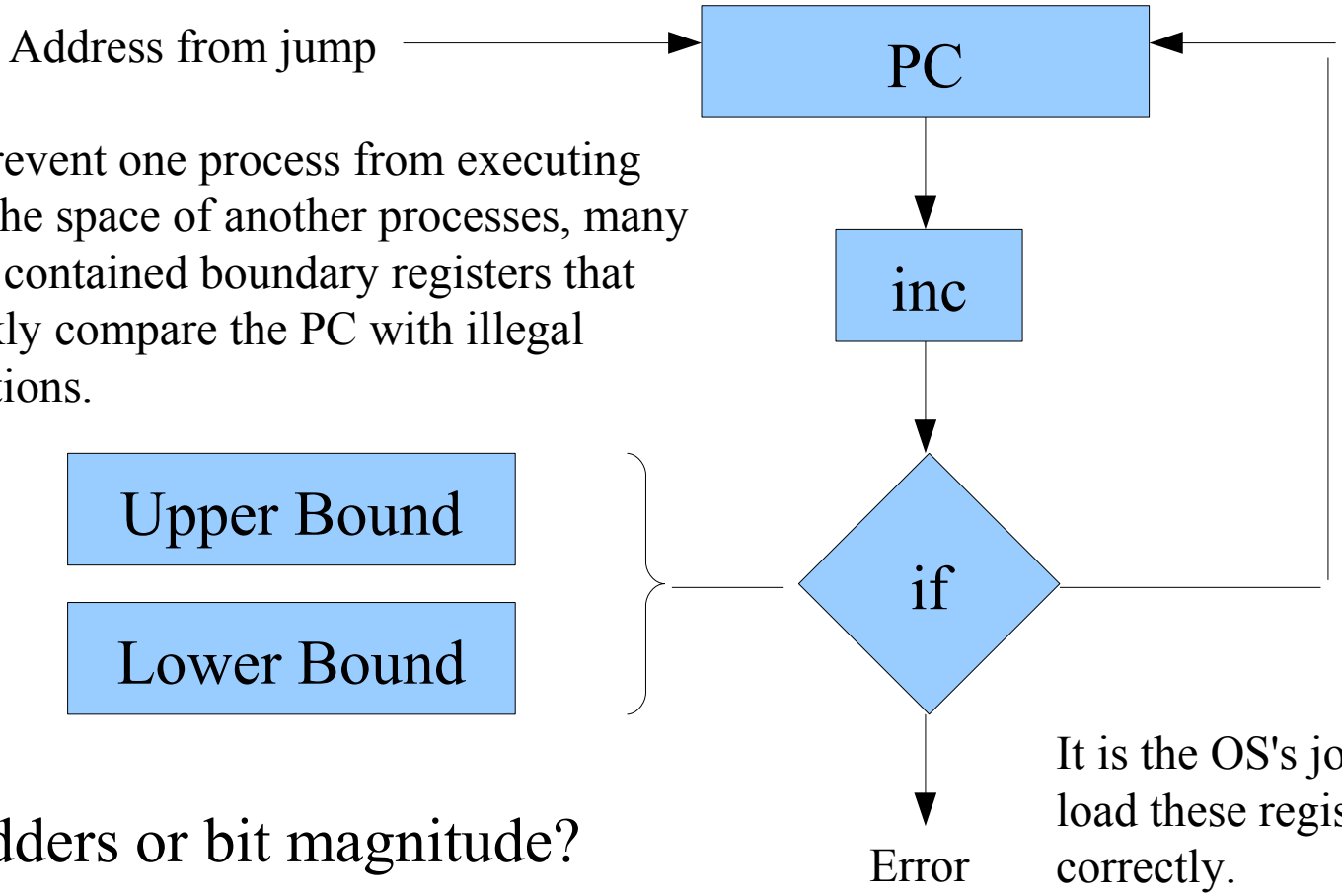
CPU

Co-processor

The CPU has the MBR and MAR to access RAM and the system bus. To access these supporting machines requires special data paths that are accessed via special instructions.

Supporting CPU Registers

Process Boundary Register



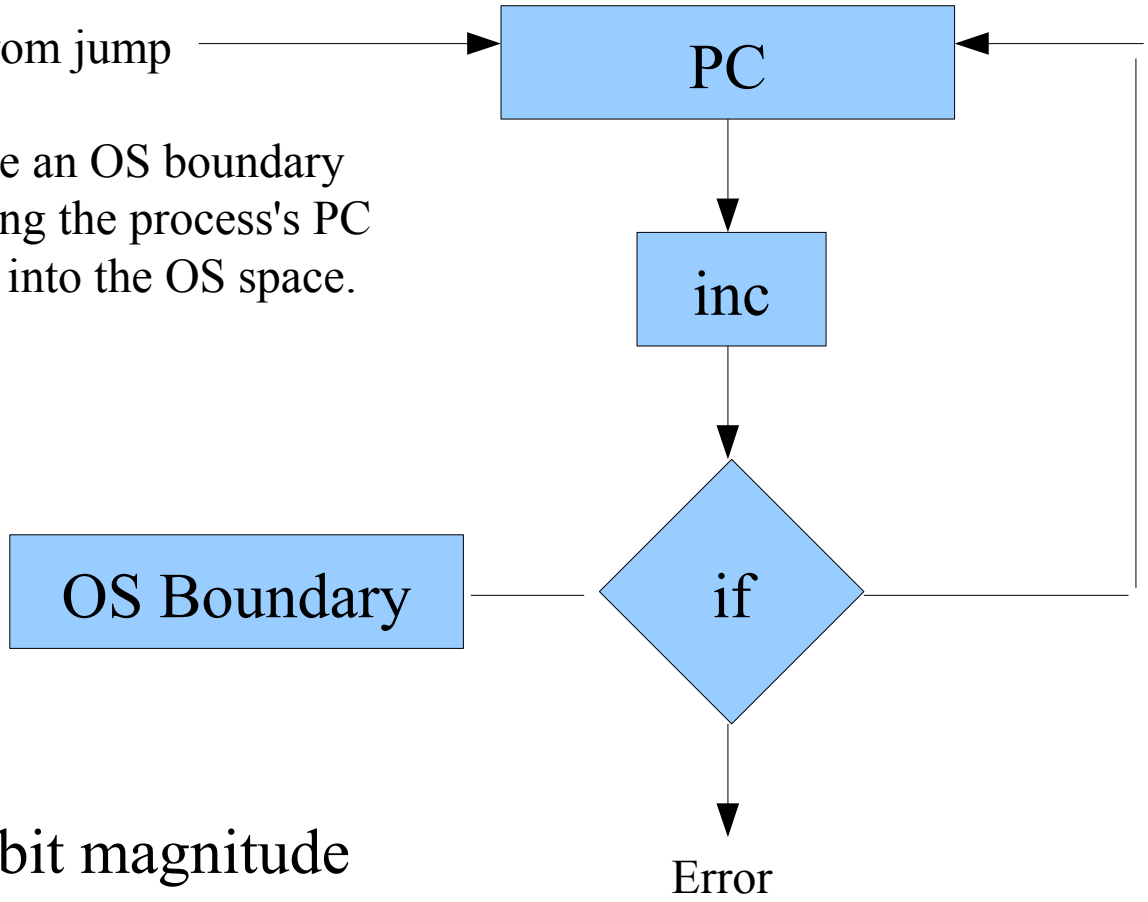
To prevent one process from executing into the space of another processes, many CPU contained boundary registers that quickly compare the PC with illegal situations.

Adders or bit magnitude?

It is the OS's job to load these registers correctly.

OS Boundary Register

Some CPUs have an OS boundary register preventing the process's PC from addressing into the OS space.

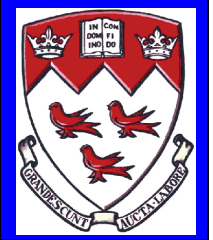


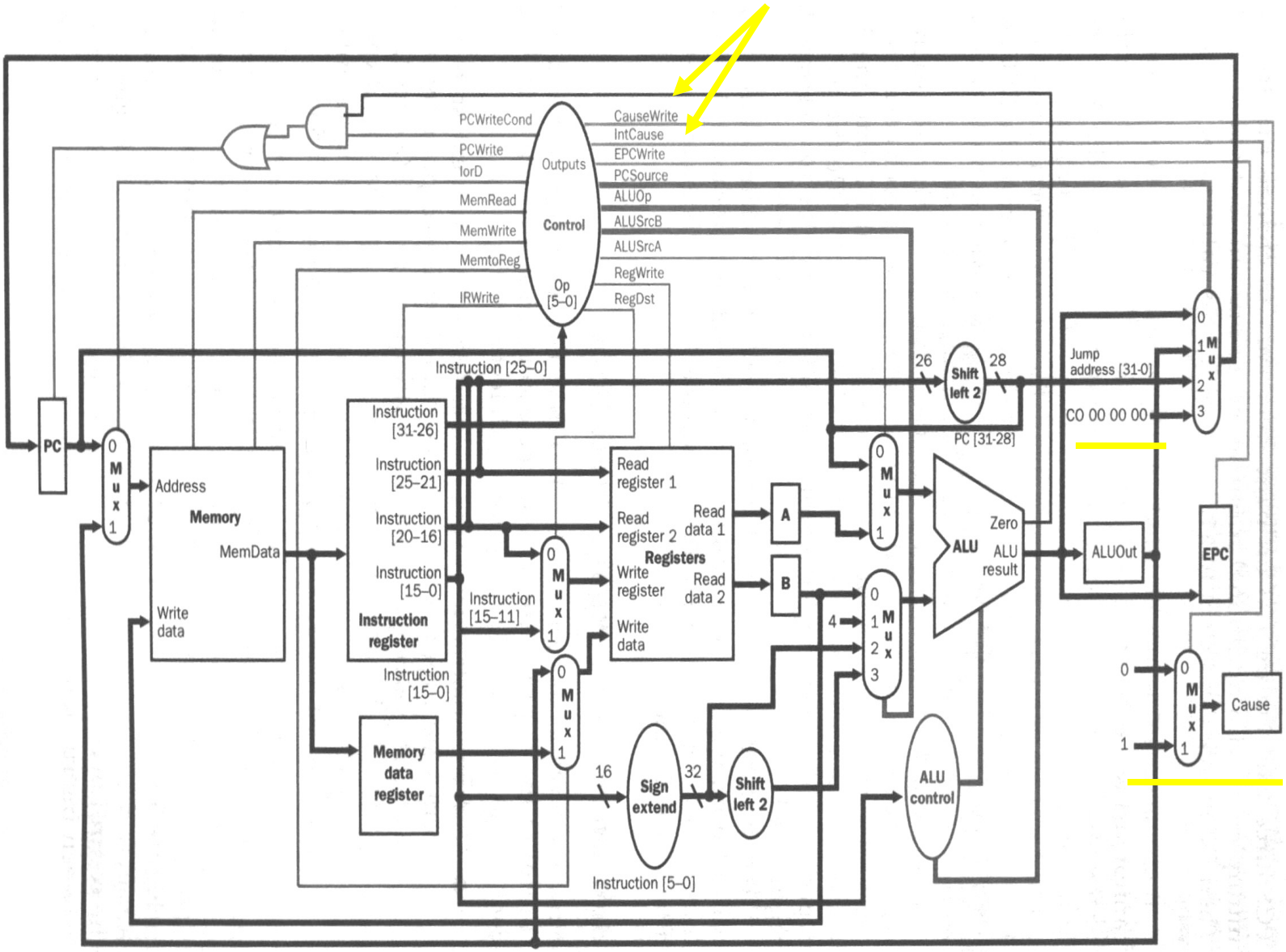
Adders or bit magnitude

Internal CPU Exception handling

- Reasons:
  - Incorrect machine language binary
  - Arithmetic: overflow, divide by zero
  - Incorrect address reference
- Supporting Registers:
  - EPC ← address of bad instruction (Exception Program Counter register)
  - Cause ← Error Code
  - Jump to reserved internal Cache memory address for exception assembler code
- Standard Exception Assembler Implementations:
  - Jump to OS exception handler vector table
  - Jump to code in user's program to handle error

COMP 273
Introduction to Computer Systems

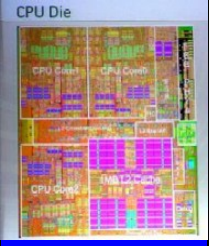




Exception Handling

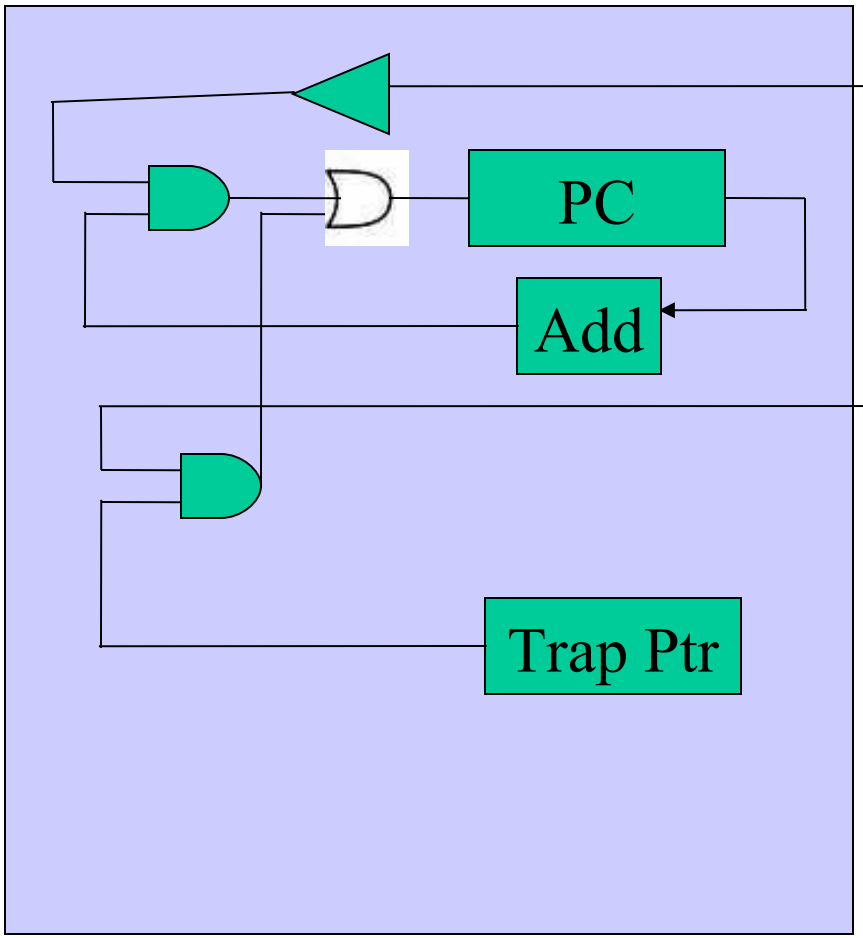
Exception handling hardware is built into the CPU with default PC address locations to interrupt execution. When an error occurs the PC is stored into EPC and then the PC is overwritten by a default address depending on the type of error. The idea is that the OS or the programmer has placed code at the address in case such an error occurs.

Supporting System-board Registers



Interrupts

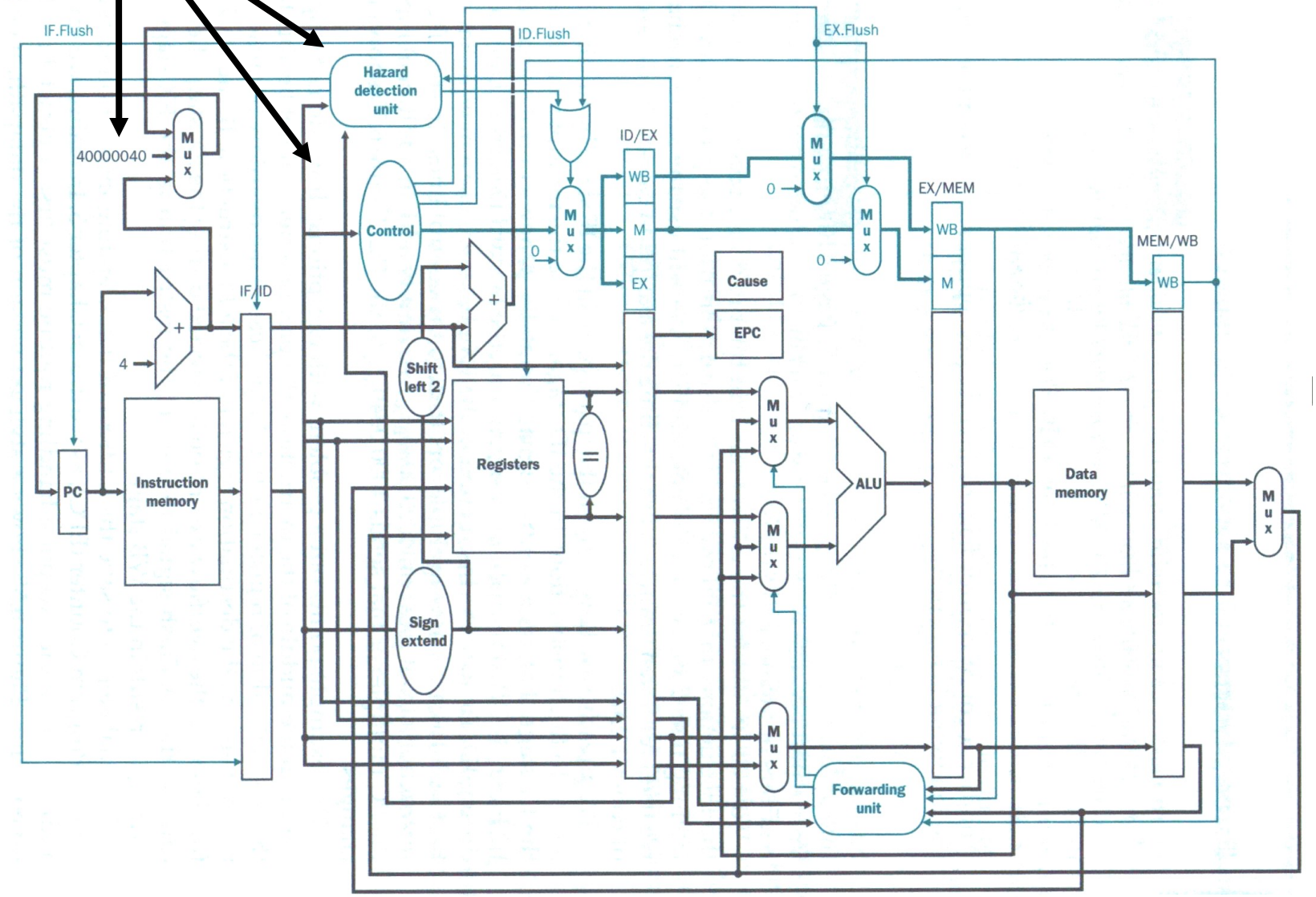
Hardware Implementation



A device needs immediate attention by the CPU. A signal is sent to overwrite or swap the PC with the Trap's ptr. In this example we see an overwrite.

Exceptions occur in a similar way except the source of the Trap signal comes from the ALU or instruction

MIPS Implementation



Part 2

Co-Processors

(Multiplication)

About Multiplication

- The CPU's ALU
  - Integer operations: + - * /
- The Co-Processor's ALU
  - Floating point operations: + - * /

Format's Radically Different

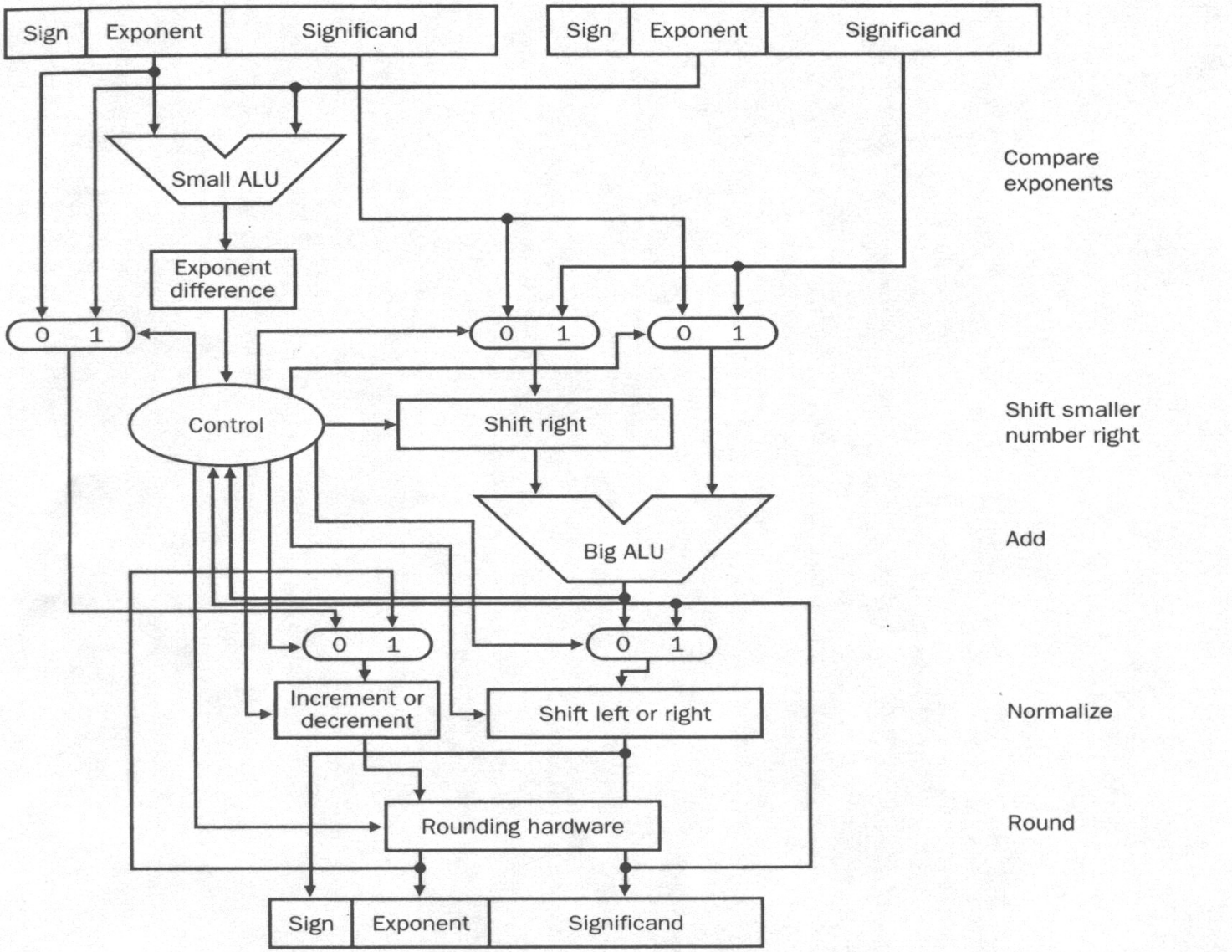
Floating point

S	exponent	mantissa

Integer

2's complement number

Floating Point Circuitry



Grade School Multiplication

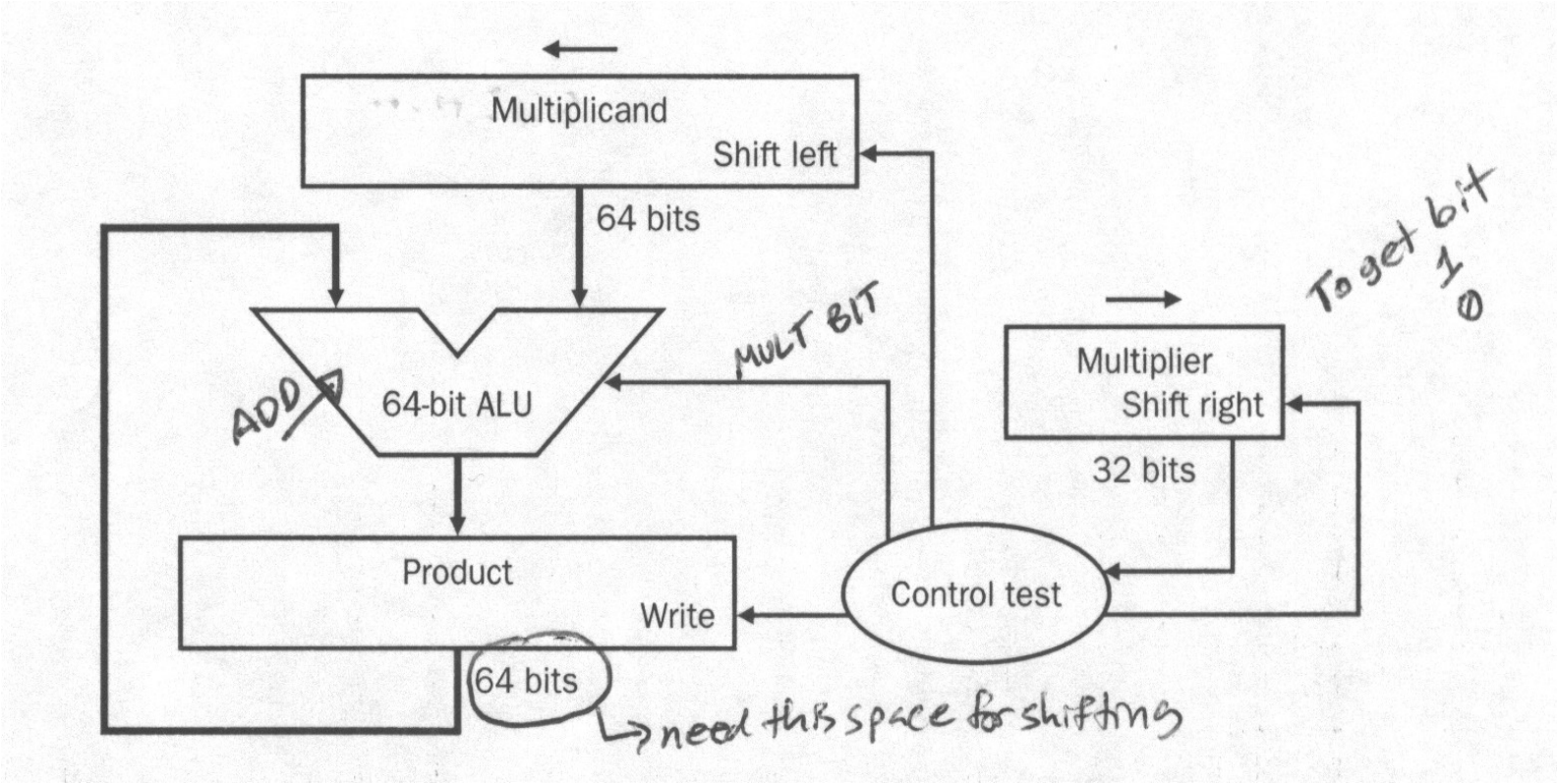
1000_{10} ← Multiplicand
$\underline{X\ 1001_{10}}$ ← Multiplier

```
   1000
  0000
 0000
1000
-------
1001000
```
$_{10}$

Step 1: right-most digit of multiplier
multiplied with multiplicand
Step 2: Write answer below digit
Step 3: Next digit of multiplier, go to step 1
Step 4: When done, sum.

Note:
- We do not need to wait to do the sum.
  The result of product is naturally shifted left
- If the multiplier is a 1 then we copy the multiplicand
- If the multiplier is a 0 then the result is zero

Integer Multiplier Hardware



- If multiplier bit is 1 then sum multiplicand with product and shift left M'd
- If multiplier bit is 0 then just shift left multiplicand

Note: MIPS does not support test if product fits in 32 bit register.

The Multiplication Procedure

Product = 2 X 3 = 0010 X 0011

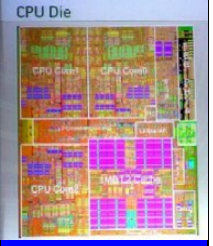
Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	001(1)	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	000(0)	0000 1000	0000 0110
3	1: 0 ⇒ no operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	000(0)	0001 0000	0000 0110
4	1: 0 ⇒ no operation	0000	0001 0000	0000 0110
Repeated for each bit	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

The circled bit is shifted out right to determine the operation

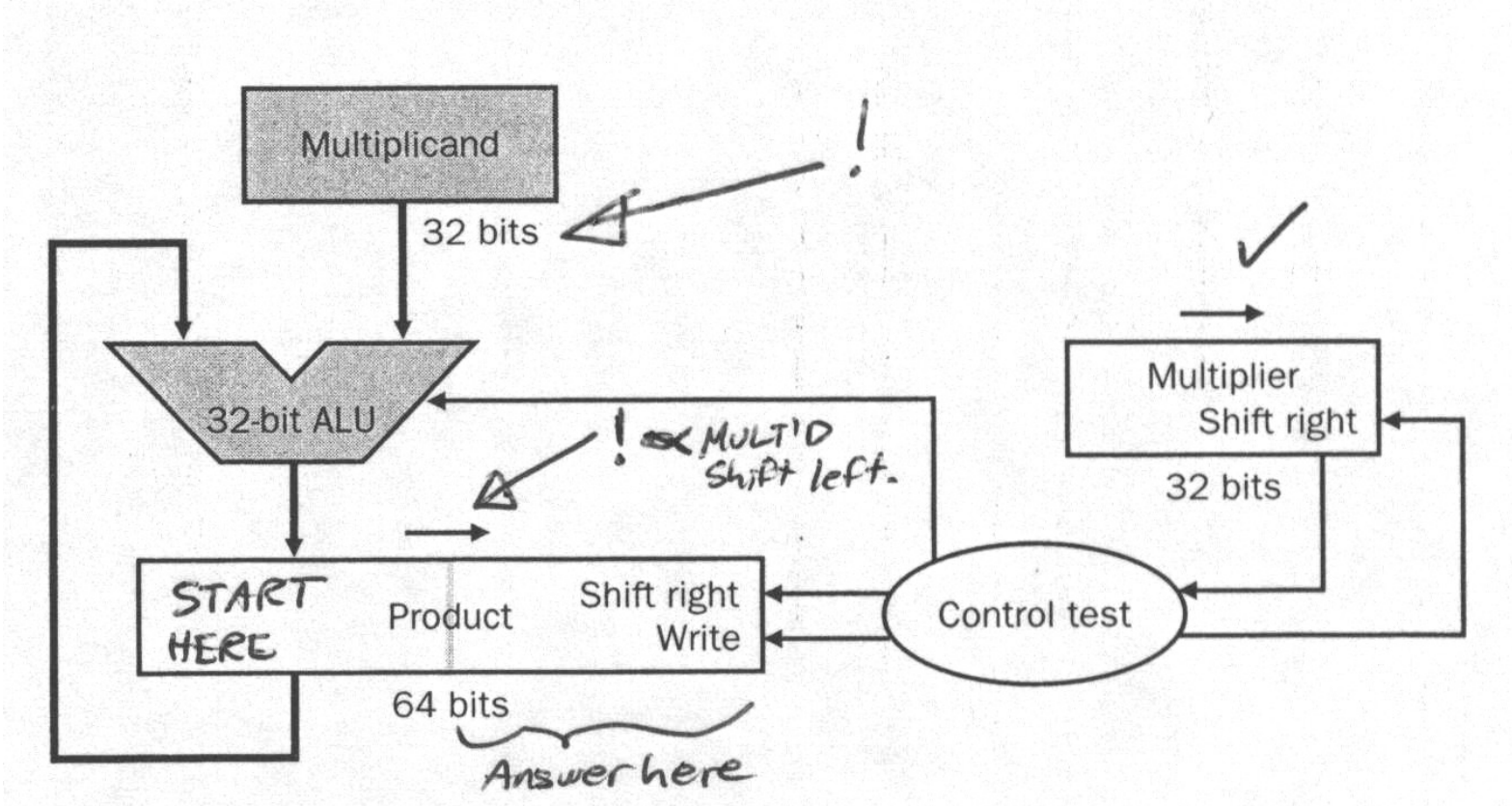
6

Notes

- Negative multiplication can be carried out by converting to a positive value then checking the signs at the end.
- Multiplications of 2^i are implemented faster using a shift left i times instruction (not using multiply).
- There is no check in MIPS for overflow

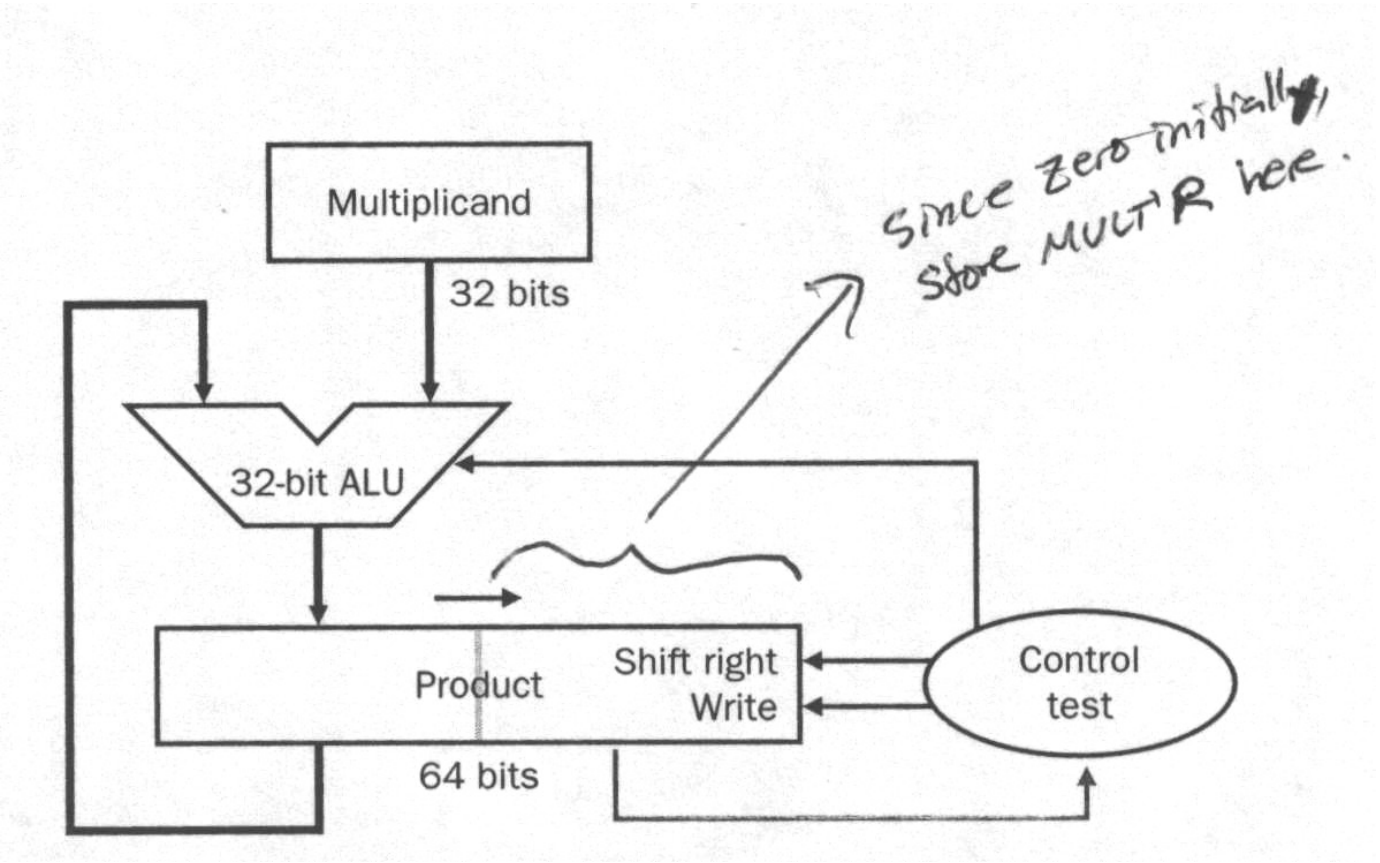


Hardware Improvement 1



Changes: Multiplicand & Multiplier are 32 bit registers
Changes: Product register is right shifted
Note : Answer in right most 32 bits

Hardware Improvement 2



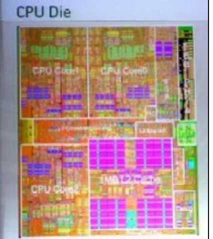
Changes: No Multiplier register!

Note:

- The multiplier is placed in the answer part of the product
- It is shifted out right as the answer is shifted in right

Part 3

Integer Division



MIPS Integer Division

- No check for divide by zero!

```
                  1001          <---- Quotient
            _________________
 1000_10  \/ 1001010_10         <---- Dividend
   ^         -1000
   |          ----
Divisor         10
                101
                1010
               -1000
                ----
                  10            <---- Remainder
```

- Find value in Div'd large enough to sub
  - Put zero in Quotient on fail
- Subtract, result 1 in Quotient & remainder
- Repeat step 1 with remainder
- Stop when no more values
  - Result: Quotient and a remainder

Another Example

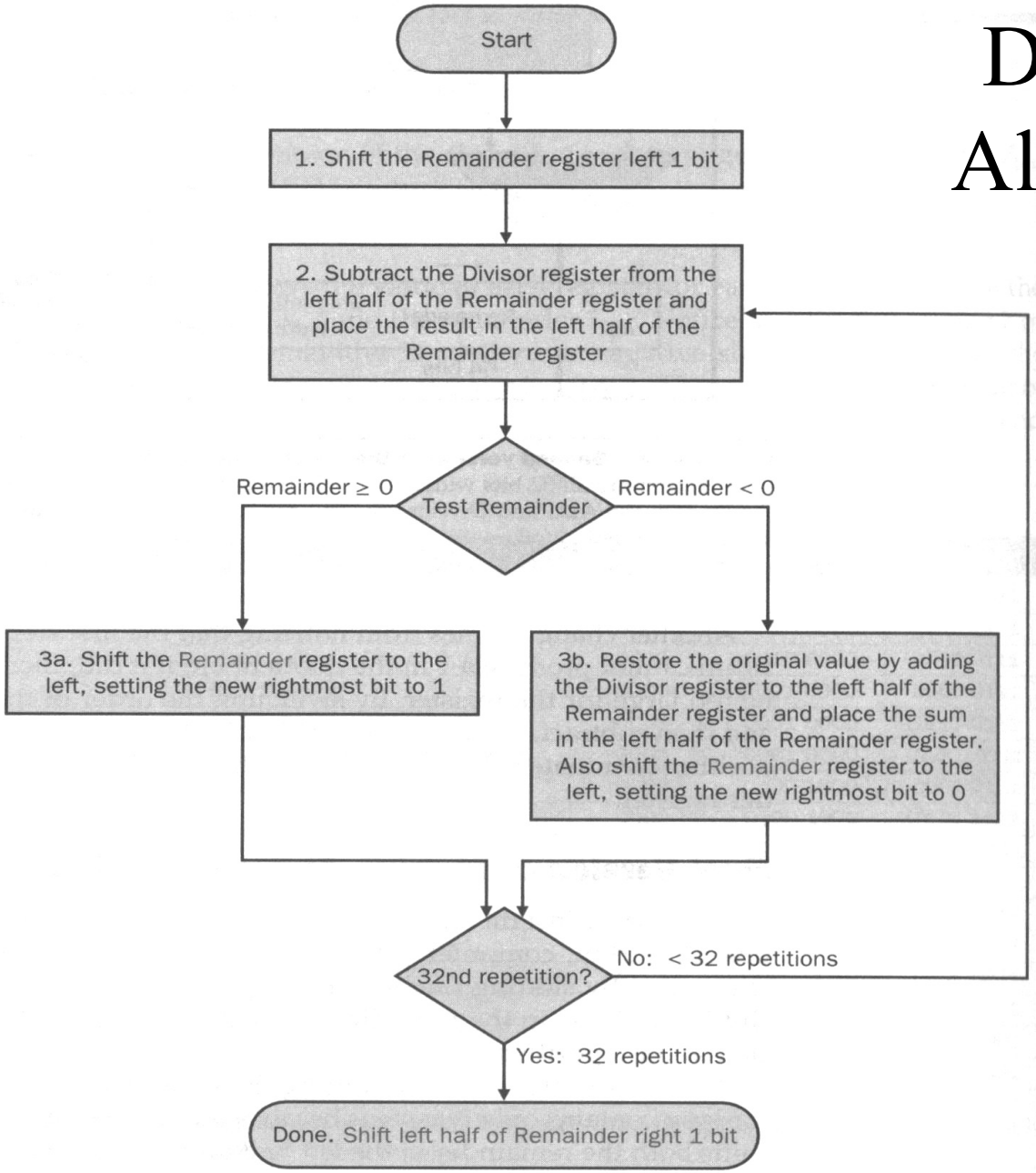
25 / 4 =

25 – 4 – 4 – 4 – 4 – 4 – 4 = 1

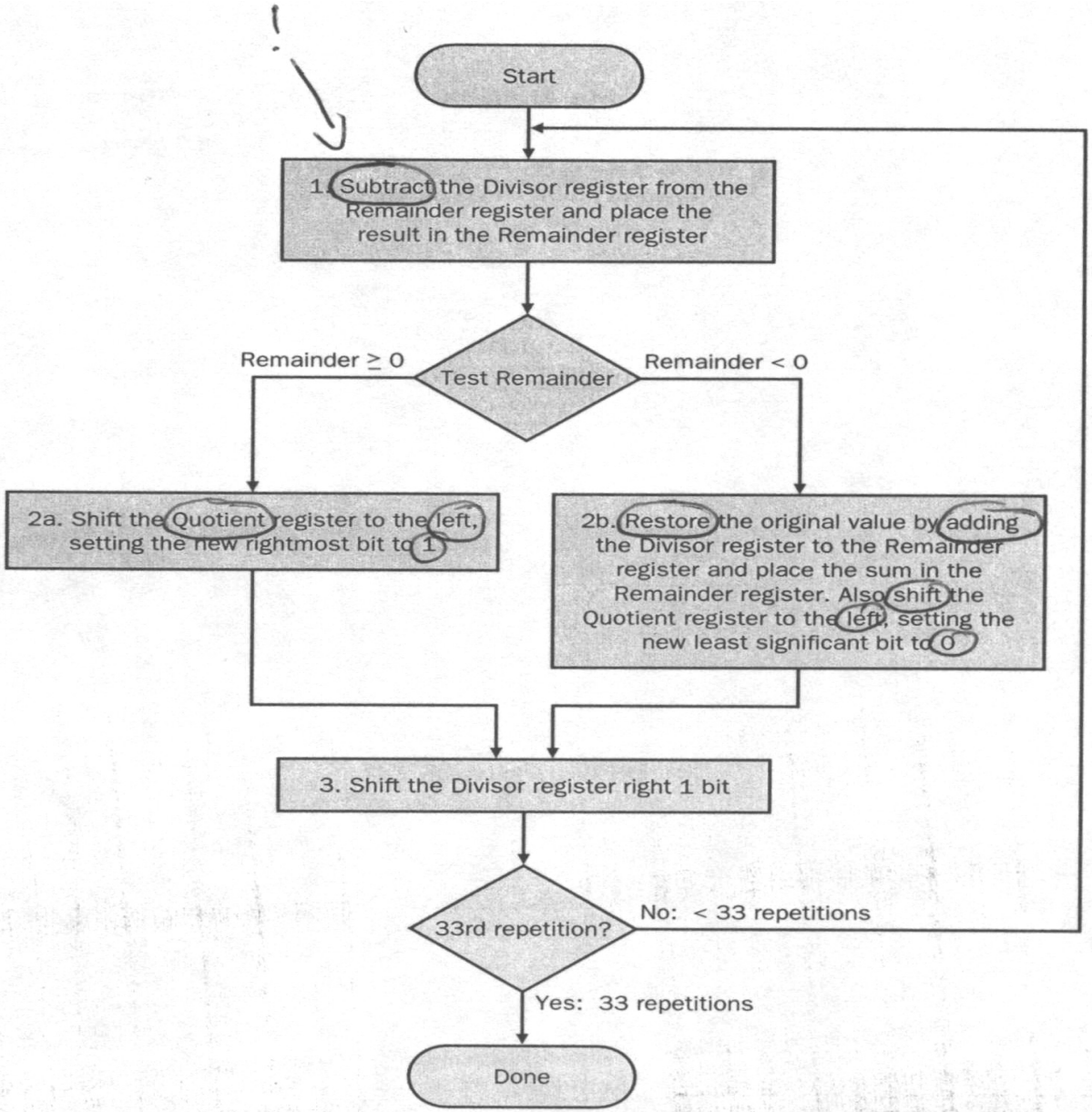
Loop & count

Remainder

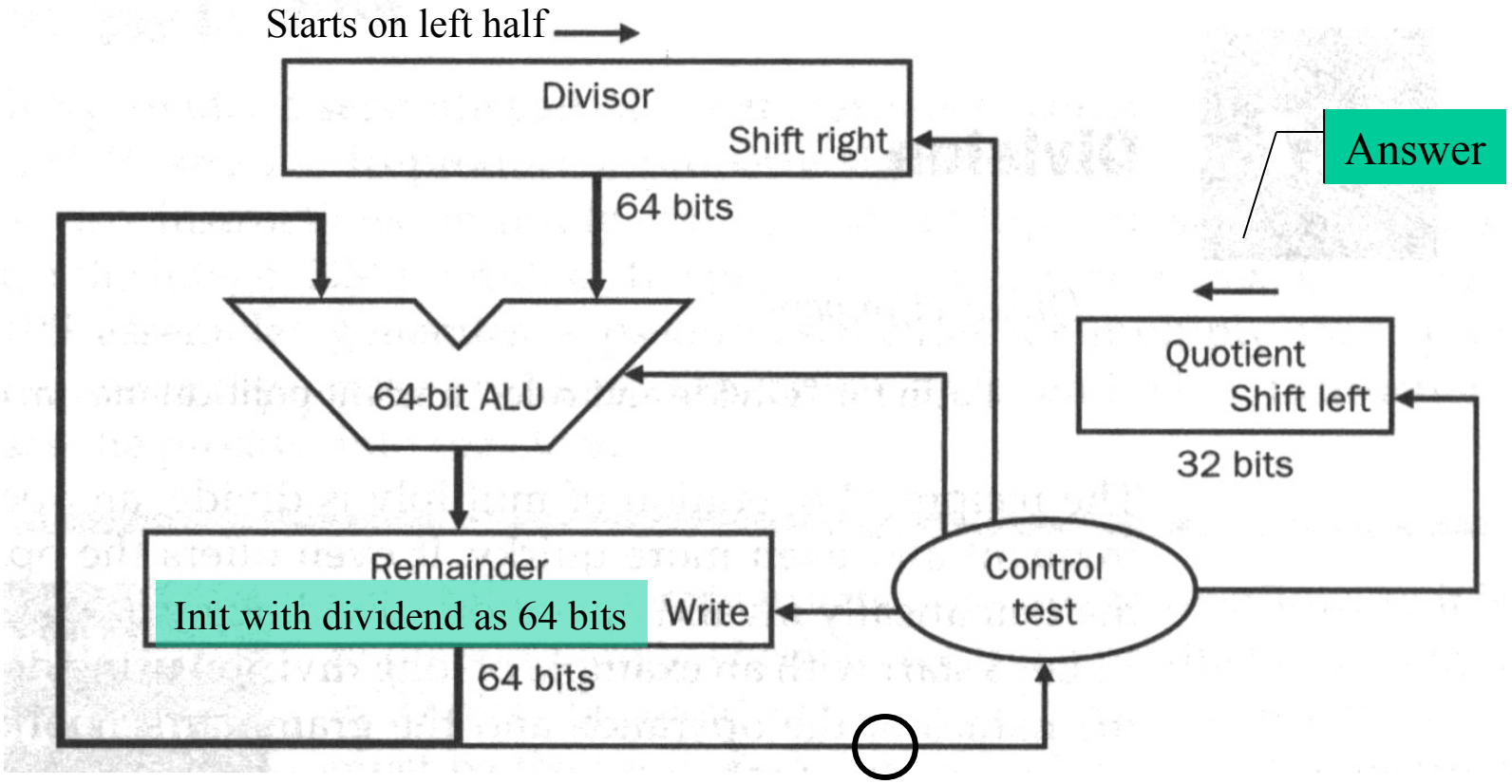
Division Algorithm



Division Flowchart



Integer Division Hardware



Control established from here

Basic Division Algorithm

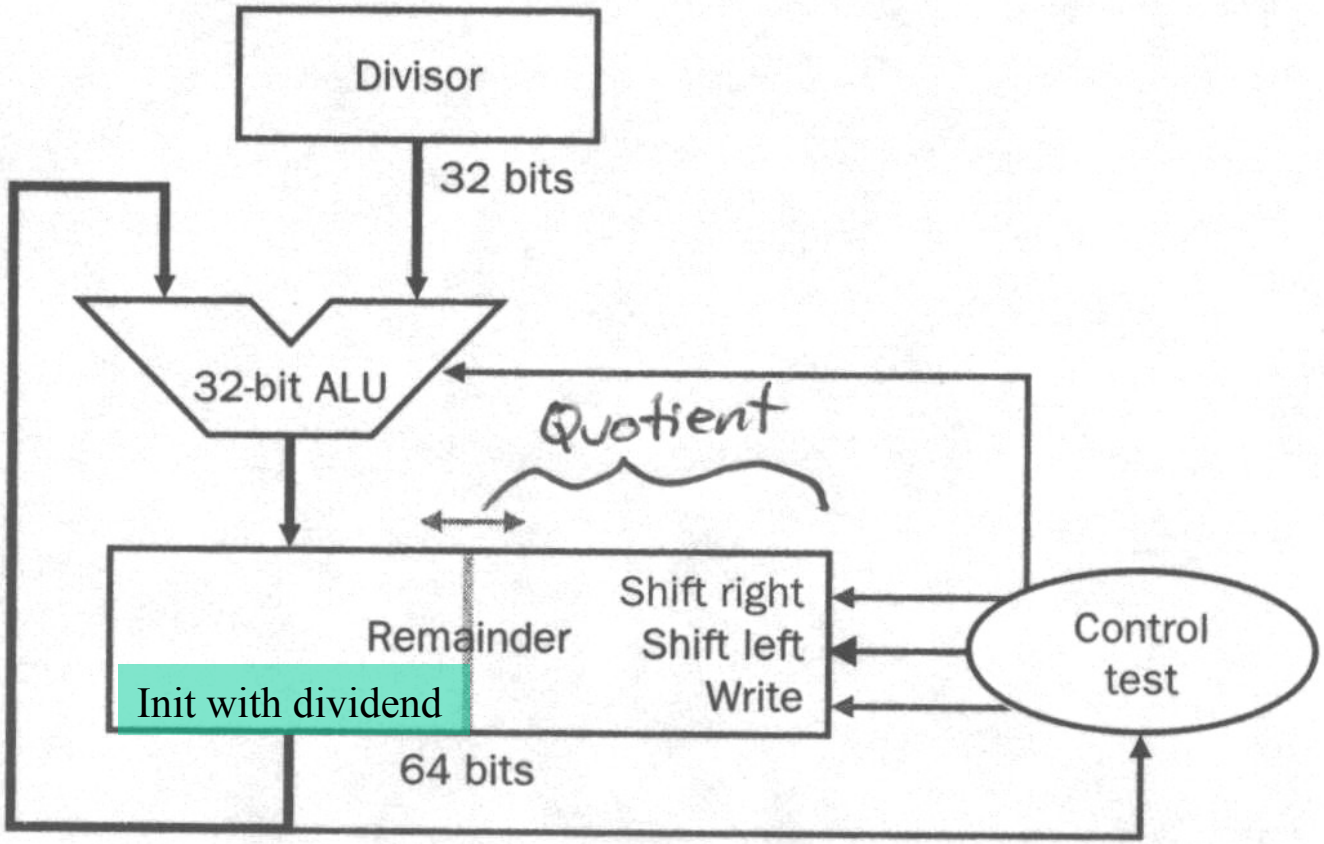
7 / 2 = 0111 / 0010 = ?

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

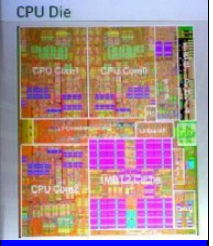
Optimization

- Shift remainder left instead of divisor right
  - Then reduce divisor register to 32 bits
- Remove Quotient register and put in right half of remainder register
- Algorithm step reduction
  - A 1 cannot be the first bit in Quotient register
  - Start with a shift of remainder left 1 to reduce loop by 1 iteration (doing that already from first modification above)

Optimized Division Hardware



Changes: Quotient register stored in right half of Remainder
Changes: Dividend is now only 32 bits
Changes: Different shift operation on each half



Division Algorithm

Iteration	Step	Divisor	Remainder
0	Initial values	0010	0000 0111
	Shift Rem left 1	0010	0000 1110
1	2: Rem = Rem – Div	0010	1110 1110
	3b: Rem < 0 ⇒ + Div, sll R, R0 = 0	0010	0001 1100
2	2: Rem = Rem – Div	0010	1111 1100
	3b: Rem < 0 ⇒ + Div, sll R, R0 = 0	0010	0011 1000
3	2: Rem = Rem – Div	0010	0001 1000
	3a: Rem ≥ 0 ⇒ sll R, R0 = 1	0010	0011 0001
4	2: Rem = Rem – Div	0010	0001 0001
	3a: Rem ≥ 0 ⇒ sll R, R0 = 1	0010	0010 0011
	Shift left half of Rem right 1	0010	0001 0011

7 / 2 = 0000 0111 / 0010

Integer Floating point Instructions

move from coprocessor register	mfc0	$s1,$epc	$s1 = $epc	Used to copy Exception PC plus other special registers
multiply	mult	$s2,$s3	Hi, Lo = $s2 × $s3	64-bit signed product in Hi, Lo
multiply unsigned	multu	$s2,$s3	Hi, Lo = $s2 × $s3	64-bit unsigned product in Hi, Lo
divide	div	$s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Lo = quotient, Hi = remainder
divide unsigned	divu	$s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Unsigned quotient and remainder
move from Hi	mfhi	$s1	$s1 = Hi	Used to get copy of Hi
move from Lo	mflo	$s1	$s1 = Lo	Used to get copy of Lo

We will explore this more when we do programming.