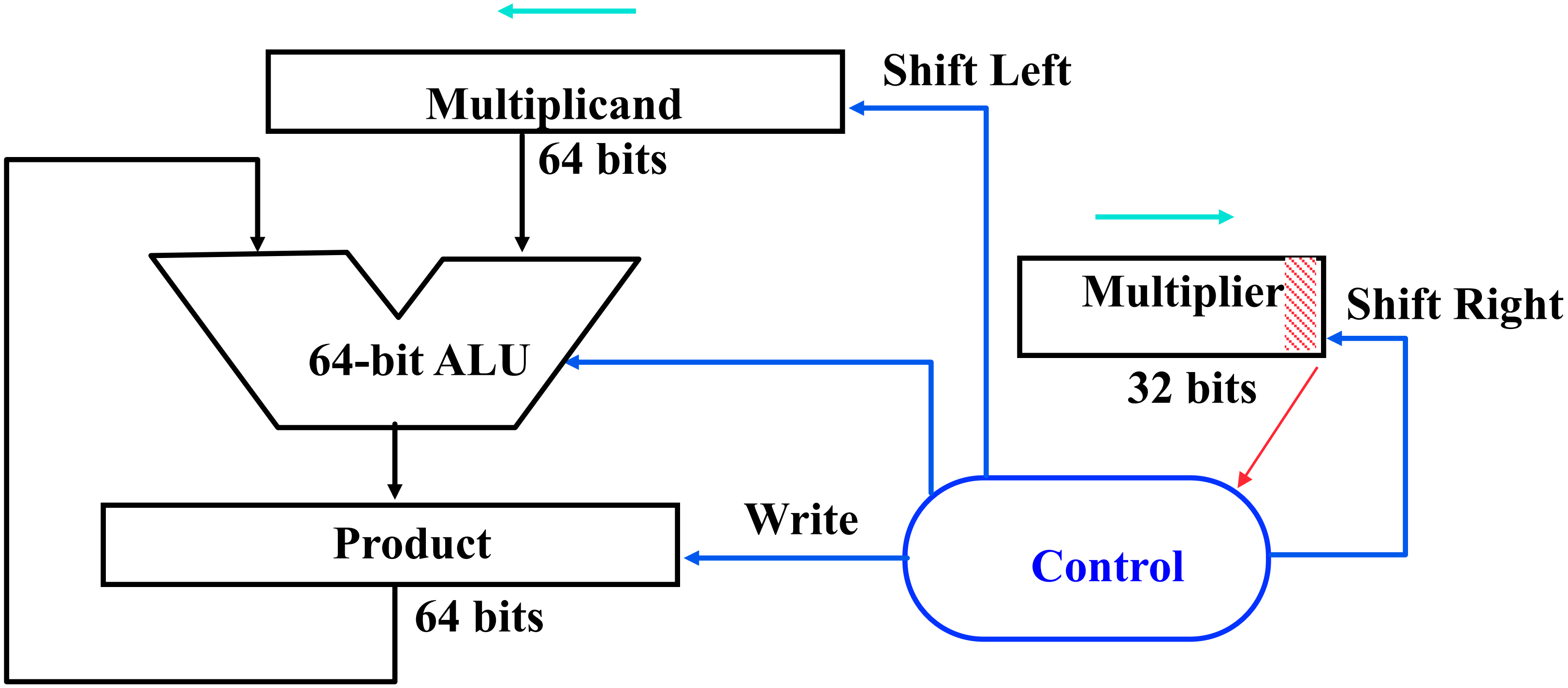
Computer Architecture
EECS 361

Lecture 7

ALU Design: Division
Designing a Single Cycle Datapath

Review: Unsigned shift-add multiplier (version 1)

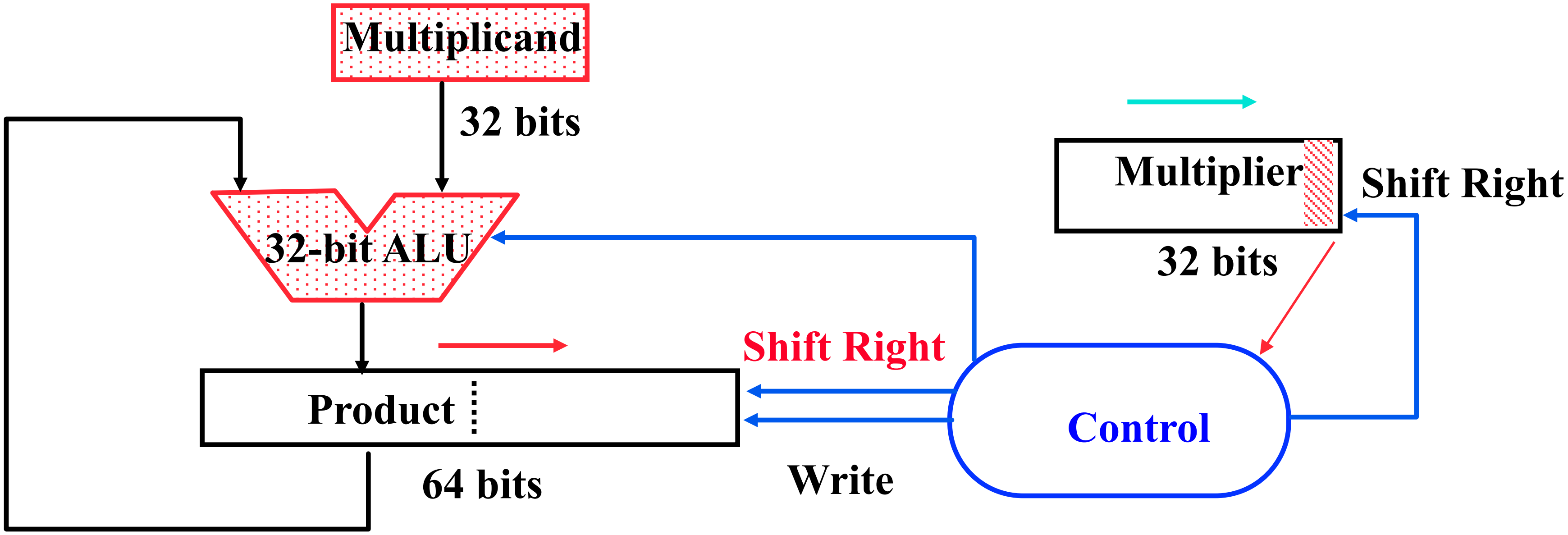
- 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit multiplier reg



Multiplier = datapath + control

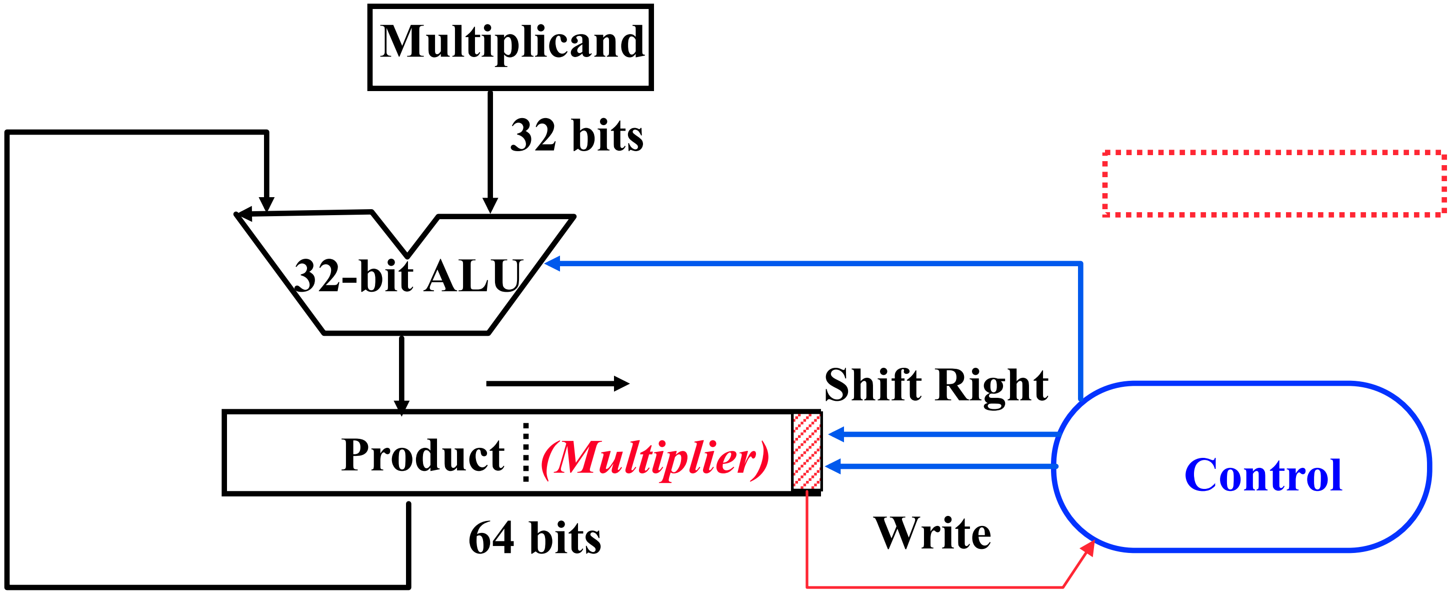
Review: Unsigned shift-add multiplier (version 2)

- 32-bit Multiplicand reg, 32 -bit ALU, 64-bit Product reg, 32-bit Multiplier reg

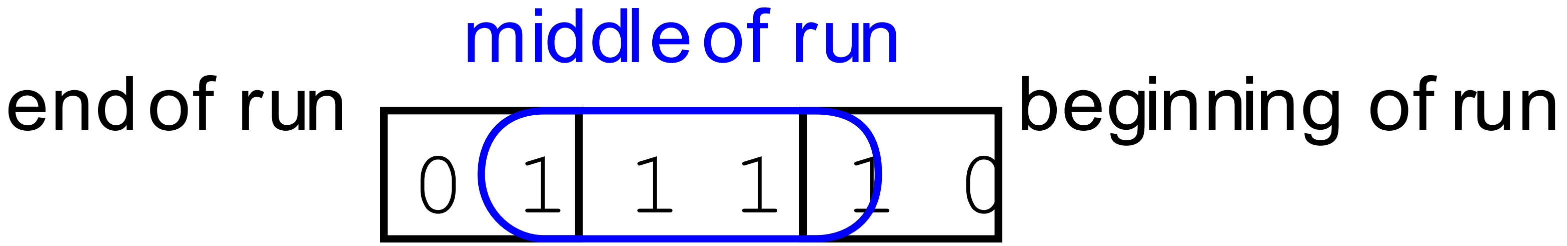


Review: Unsigned shift-add multiplier (version 3)

- 32-bit Multiplicand reg, 32 -bit ALU, 64-bit Product reg, (0-bit Multiplier reg)



Review: Booth’s Algorithm



Current Bit	Bit to the Right	Explanation	Example
1	0	Beginning of a run of 1s	0001111000
1	1	Middle of a run of 1s	0001111000
0	1	End of a run of 1s	0001111000
0	0	Middle of a run of 0s	0001111000

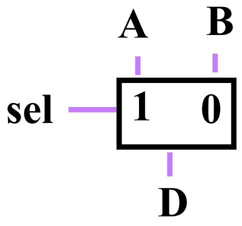
Originally for Speed since shift faster than add for his machine

$$2^n - 2^k = 00000111111111100000$$

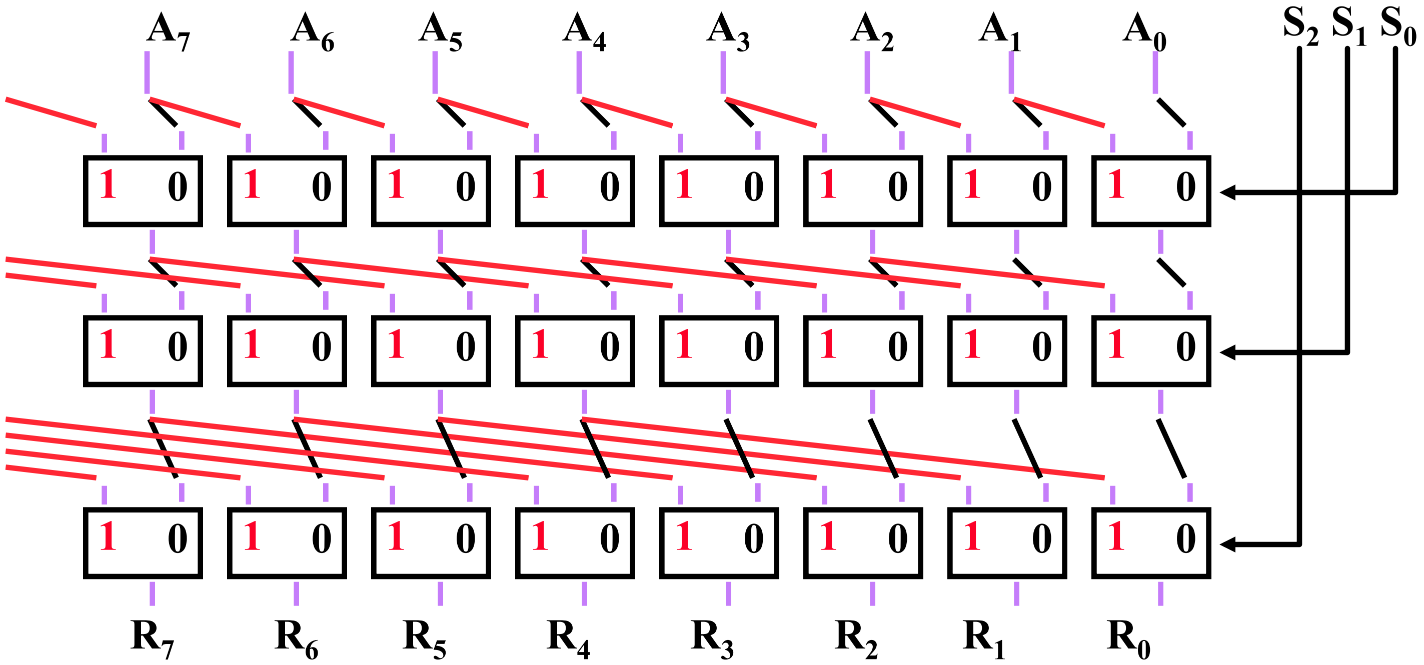
n ……….. k ……. 0 (bit position)

Review: Combinational Shifter from MUXes

Basic Building Block



8-bit right shifter



- What comes in the MSBs?
- How many levels for 32-bit shifter?
- What if we use 4-1 Muxes ?

Outline of Today’s Lecture

- Divide
- Introduction to Single cycle processor design

Divide: Paper & Pencil

```
                  1001     Quotient
             _________
Divisor 1000 | 1001010     Dividend
              -1000
                  10
                  101
                  1010
                 -1000
                    10     Remainder (or Modulo result)
```

See how big a number can be subtracted, creating quotient bit on each step

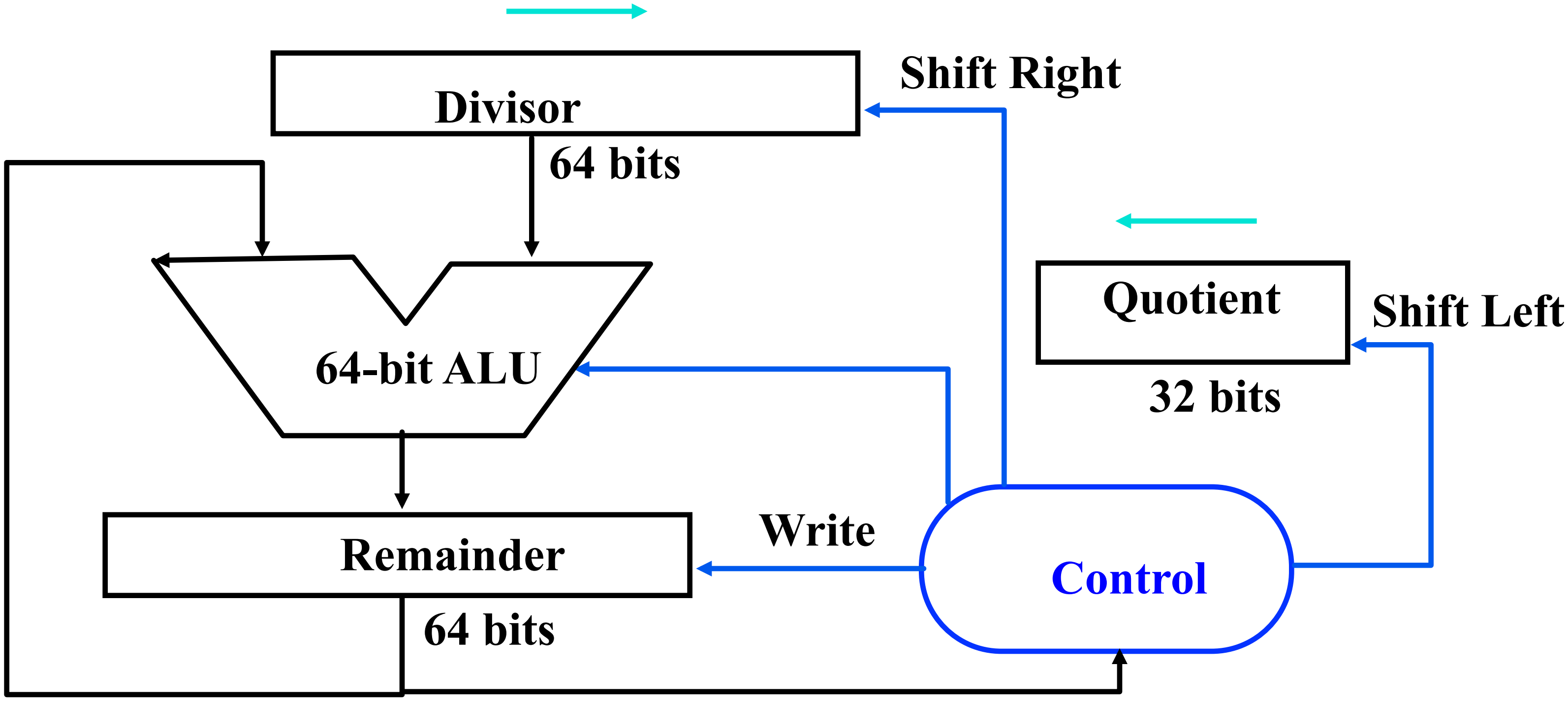
Binary => 1 * divisor or 0 * divisor

Dividend = Quotient x Divisor + Remainder
=> | Dividend | = | Quotient | + | Divisor |

3 versions of divide, successive refinement

DIVIDE HARDWARE Version 1

- 64-bit Divisor reg, 64-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg

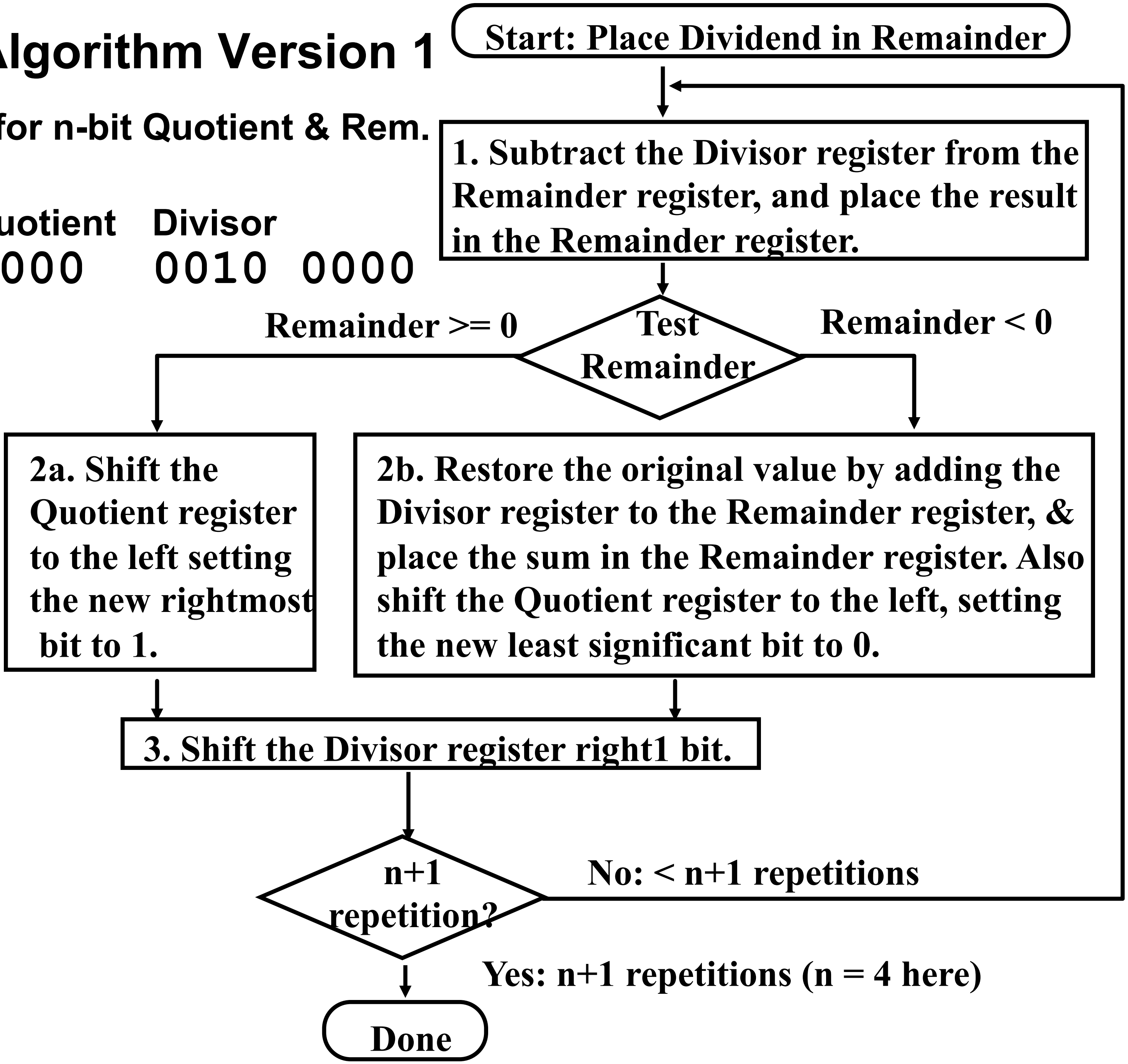


Divide Algorithm Version 1

°Takes n+1 steps for n-bit Quotient & Rem.

Remainder **Quotient** **Divisor**

```
0000  0111 0000   0010  0000
```



Start: Place Dividend in Remainder

1. Subtract the Divisor register from the Remainder register, and place the result in the Remainder register.

Test Remainder

Remainder >= 0

Remainder < 0

2a. Shift the Quotient register to the left setting the new rightmost bit to 1.

2b. Restore the original value by adding the Divisor register to the Remainder register, & place the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0.

3. Shift the Divisor register right1 bit.

n+1 repetition?

No: < n+1 repetitions

Yes: n+1 repetitions (n = 4 here)

Done

Observations on Divide Version 1

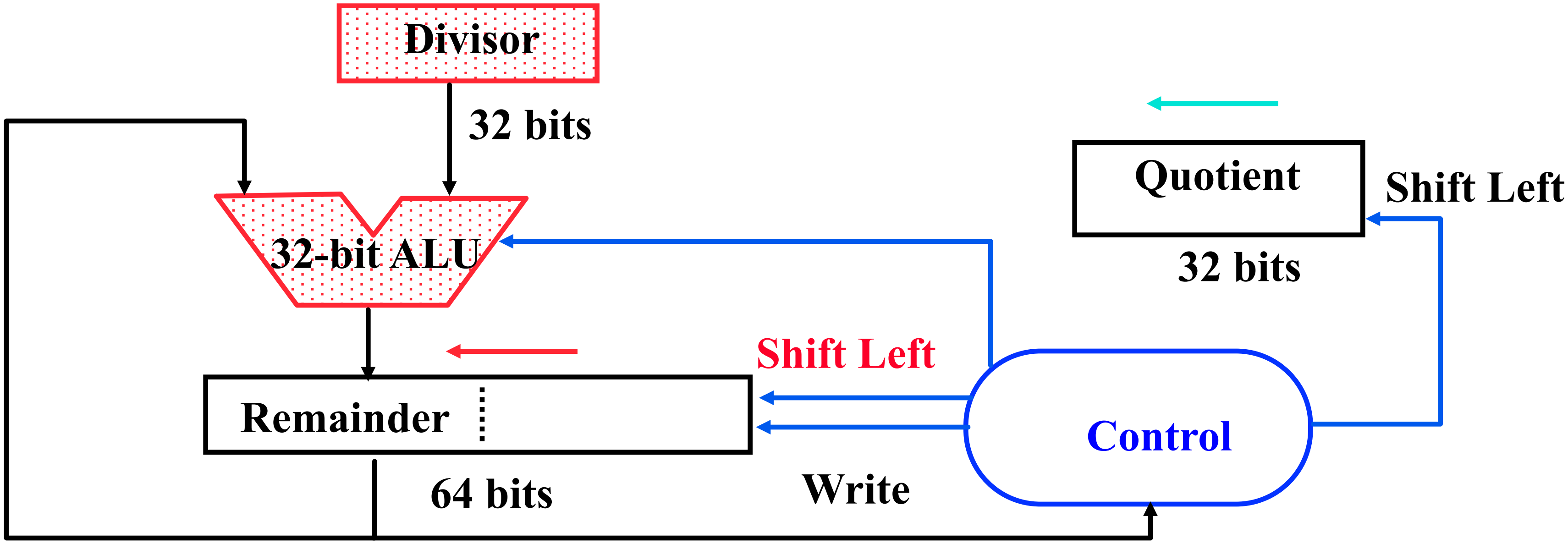
- 1/2 bits in divisor always 0
  => 1/2 of 64-bit adder is wasted
  => 1/2 of divisor is wasted

- Instead of shifting divisor to right,
  shift remainder to left?

- 1st step cannot produce a 1 in quotient bit
  (otherwise too big)
  => switch order to shift first and then subtract,
  can save 1 iteration

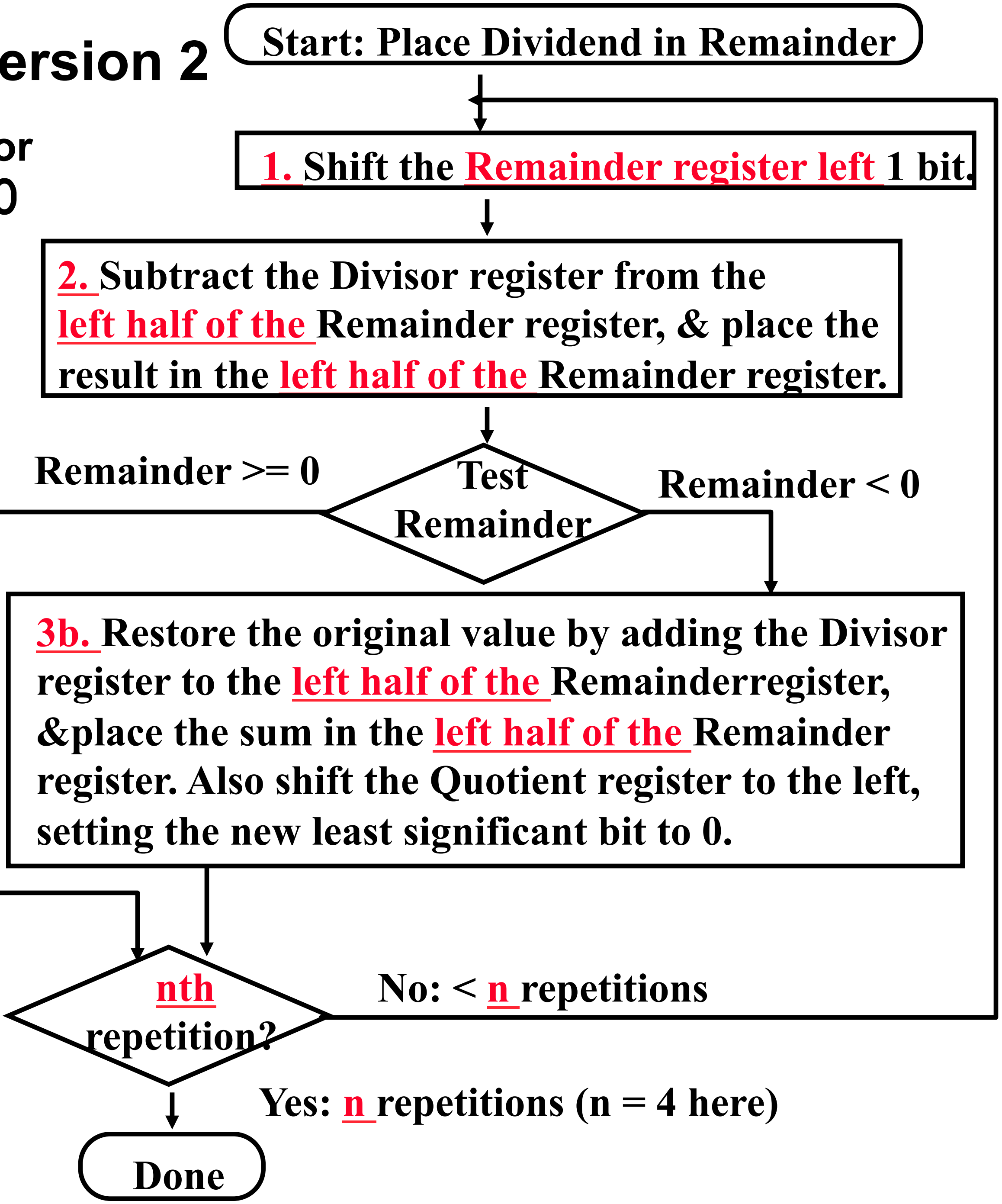
DIVIDE HARDWARE Version 2

- 32-bit Divisor reg, 32-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg



Divide Algorithm Version 2

Remainder	Quotient	Divisor
0000 0111	0000	0010



Start: Place Dividend in Remainder

1. Shift the Remainder register left 1 bit.

2. Subtract the Divisor register from the left half of the Remainder register, & place the result in the left half of the Remainder register.

Test Remainder

Remainder >= 0

Remainder < 0

3a. Shift the Quotient register to the left setting the new rightmost bit to 1.

3b. Restore the original value by adding the Divisor register to the left half of the Remainderregister, &place the sum in the left half of the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0.

nth repetition?

No: < n repetitions

Yes: n repetitions (n = 4 here)

Done

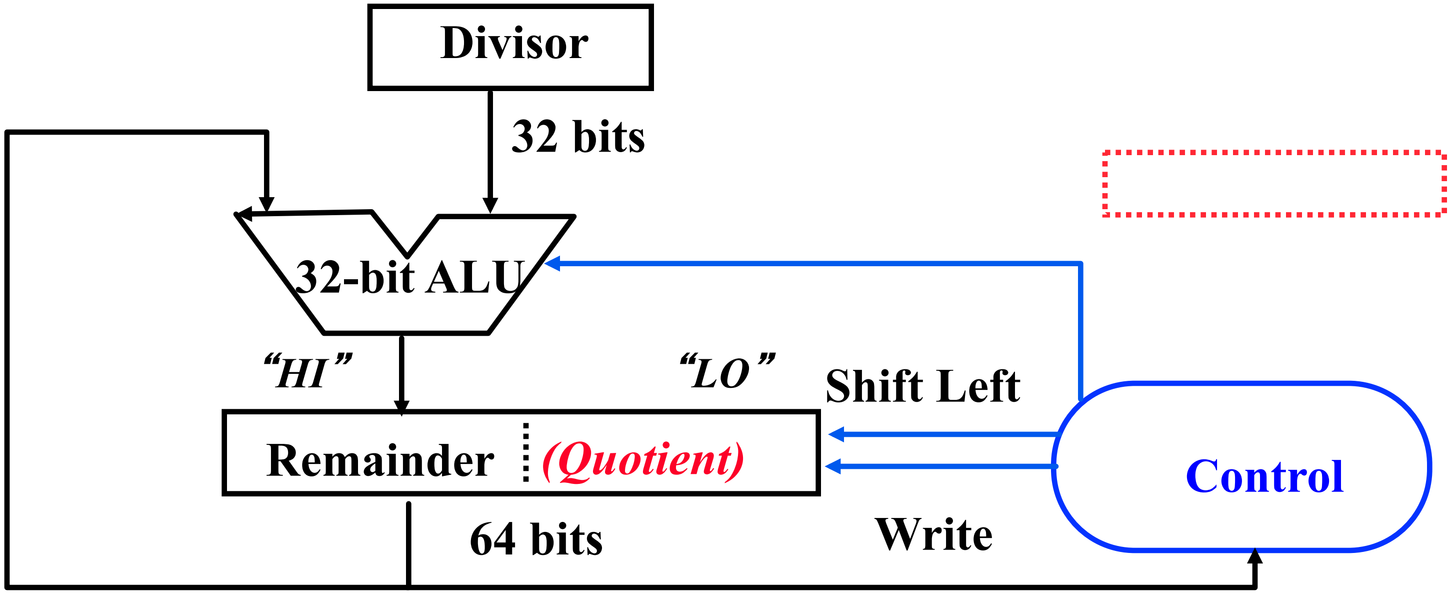
Observations on Divide Version 2

° Eliminate Quotient register by combining with Remainder as shifted left

- Start by shifting the Remainder left as before.
- Thereafter loop contains only two steps because the shifting of the Remainder register shifts both the remainder in the left half and the quotient in the right half
- The consequence of combining the two registers together and the new order of the operations in the loop is that the remainder will be shifted left one time too many.
- Thus the final correction step must shift back only the remainder in the left half of the register

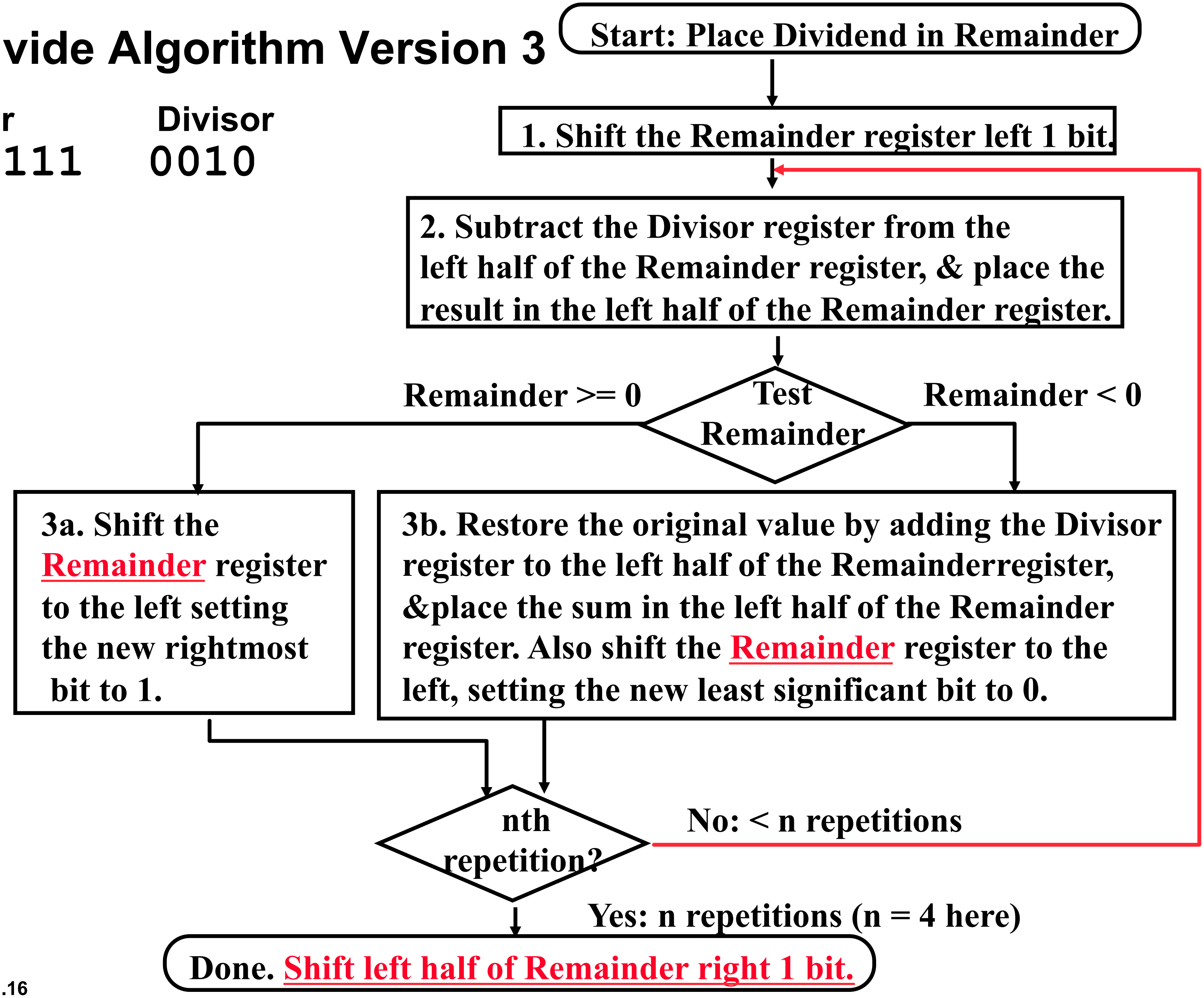
DIVIDE HARDWARE Version 3

- 32-bit Divisor reg, 32 -bit ALU, 64-bit Remainder reg, (0-bit Quotient reg)



Divide Algorithm Version 3

Remainder	Divisor
0000 0111	0010



Start: Place Dividend in Remainder

1. Shift the Remainder register left 1 bit.

2. Subtract the Divisor register from the left half of the Remainder register, & place the result in the left half of the Remainder register.

Test Remainder

Remainder >= 0

Remainder < 0

3a. Shift the Remainder register to the left setting the new rightmost bit to 1.

3b. Restore the original value by adding the Divisor register to the left half of the Remainderregister, &place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new least significant bit to 0.

nth repetition?

No: < n repetitions

Yes: n repetitions (n = 4 here)

Done. Shift left half of Remainder right 1 bit.

Observations on Divide Version 3

- Same Hardware as Multiply: just need ALU to add or subtract, and 63-bit register to shift left or shift right

- Hi and Lo registers in MIPS combine to act as 64-bit register for multiply and divide

- Signed Divides: Simplest is to remember signs, make positive, and complement quotient and remainder if necessary
  - Note: Dividend and Remainder must have same sign
  - Note: Quotient negated if Divisor sign & Dividend sign disagree e.g., –7 ÷ 2 = –3, remainder = –1

- Possible for quotient to be too large: if divide 64-bit integer by 1, quotient is 64 bits (“called saturation”)

Summary

- Bits have no inherent meaning: operations determine whether they are really ASCII characters, integers, floating point numbers
- Divide can use same hardware as multiply: Hi & Lo registers in MIPS

Designing a Single Cycle Datapath

Outline

° Where are we with respect to the BIG picture?

° The Steps of Designing a Processor

° Datapath and timing for Reg-Reg Operations

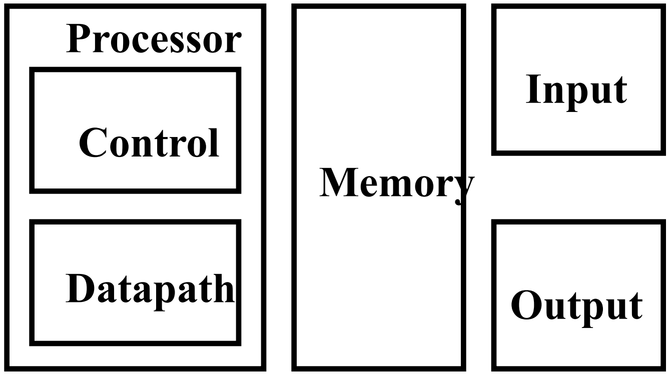
° Datapath for Logical Operations with Immediate

° Datapath for Load and Store Operations

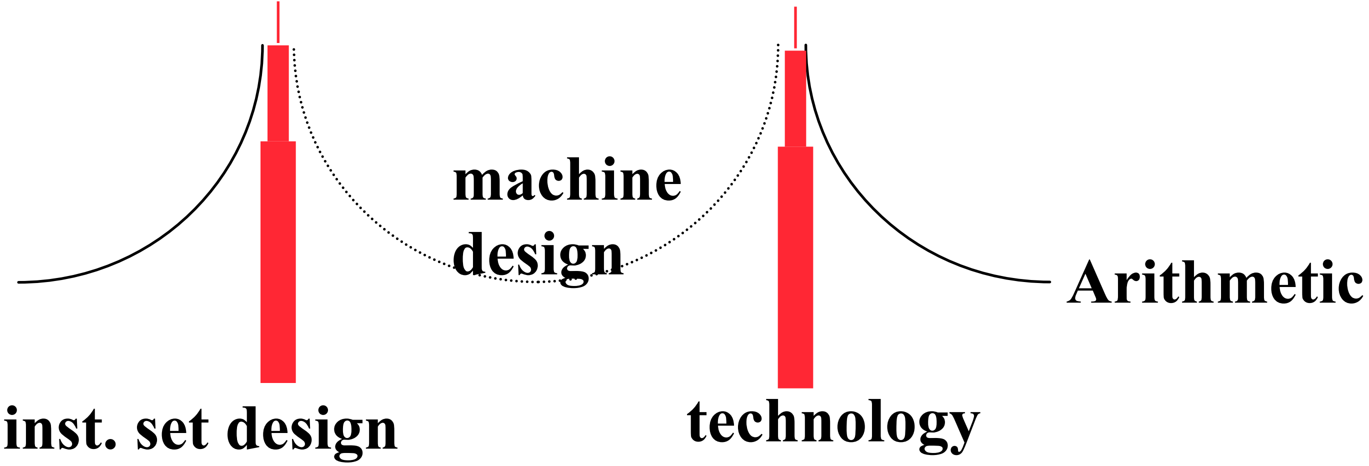
° Datapath for Branch and Jump Operations

The Big Picture: Where are We Now?

- The Five Classic Components of a Computer

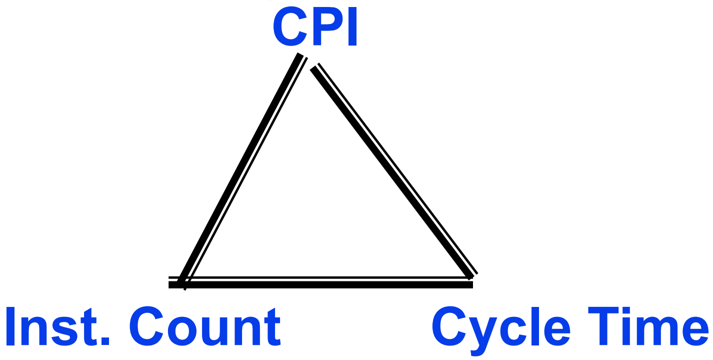


- Next Topic: Design a Single Cycle Processor



The Big Picture: The Performance Perspective

- Performance of a machine is determined by:
  - Instruction count
  - Clock cycle time
  - Clock cycles per instruction



- Processor design (datapath and control) will determine:
  - Clock cycle time
  - Clock cycles per instruction

- Single cycle processor:
  - Advantage: One clock cycle per instruction
  - Disadvantage: long cycle time

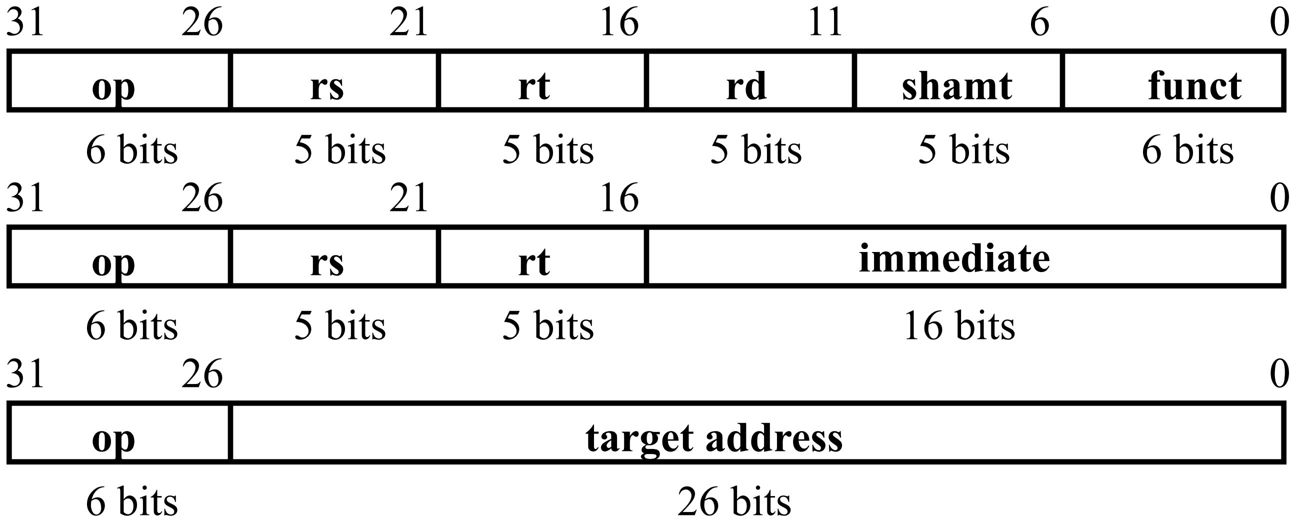
How to Design a Processor: step-by-step

- 1. Analyze instruction set => datapath requirements
  - the meaning of each instruction is given by the *register transfers*
  - datapath must include storage element for ISA registers
    - possibly more
  - datapath must support each register transfer
- 2. Select set of datapath components and establish clocking methodology
- 3. Assemble datapath meeting the requirements
- 4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
- 5. Assemble the control logic

The MIPS Instruction Formats

° All MIPS instructions are 32 bits long. The three instruction formats:

- R-type
- I-type
- J-type



R-type

31 26 21 16 11 6 0

op | rs | rt | rd | shamt | funct

6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits

I-type

31 26 21 16 0

op | rs | rt | immediate

6 bits | 5 bits | 5 bits | 16 bits

J-type

31 26 0

op | target address

6 bits | 26 bits

° The different fields are:

- op: operation of the instruction
- rs, rt, rd: the source and destination register specifiers
- shamt: shift amount
- funct: selects the variant of the operation in the “op” field
- address / immediate: address offset or immediate value
- target address: target address of the jump instruction

Step 1a: The MIPS-lite Subset for today

- ADD and SUB
  - addU rd, rs, rt
  - subU rd, rs, rt

31 26 21 16 11 6 0

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- OR Immediate:
  - ori rt, rs, imm16

31 26 21 16 0

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

- LOAD and STORE Word
  - lw rt, rs, imm16
  - sw rt, rs, imm16

31 26 21 16 0

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

- BRANCH:
  - beq rs, rt, imm16

31 26 21 16 0

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

Logical Register Transfers

- ° RTL gives the meaning of the instructions

- ° All start by fetching the instruction

op | rs | rt | rd | shamt | funct = MEM[PC]

op | rs | rt | Imm16 = MEM[PC]

inst	Register Transfers	
ADDU	R[rd] <– R[rs] + R[rt];	PC <– PC + 4
SUBU	R[rd] <– R[rs] – R[rt];	PC <– PC + 4
ORi	R[rt] <– R[rs] + zero_ext(Imm16);	PC <– PC + 4
LOAD	R[rt] <– MEM[R[rs] + sign_ext(Imm16)];	PC <– PC + 4
STORE	MEM[R[rs] + sign_ext(Imm16)] <– R[rt];	PC <– PC + 4

BEQ	if (R[rs] == R[rt]) then PC <– PC + 4 + sign_ext(Imm16)]
	else PC <– PC + 4	

Step 1: Requirements of the Instruction Set

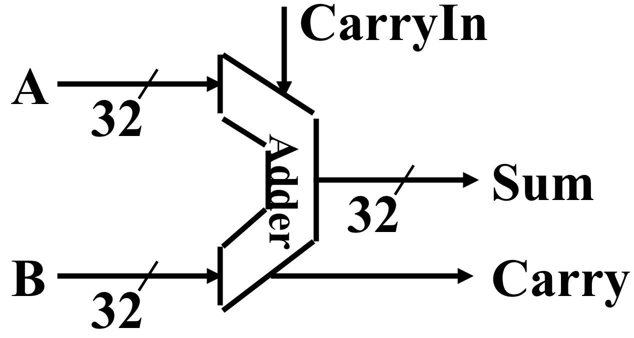
- Memory
  - instruction & data
- Registers (32 x 32)
  - read RS
  - read RT
  - Write RT or RD
- PC
- Other elements to complete datapath

Step 2: Components of the Datapath

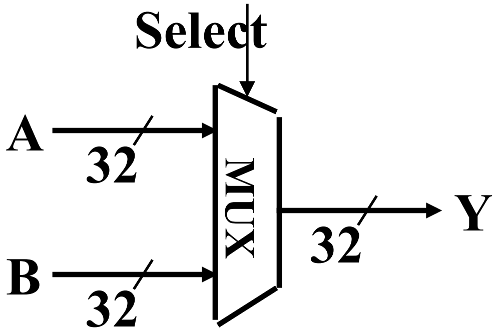
- Combinational Elements
- Storage Elements
  - Clocking methodology

Combinational Logic Elements (Basic Building Blocks)

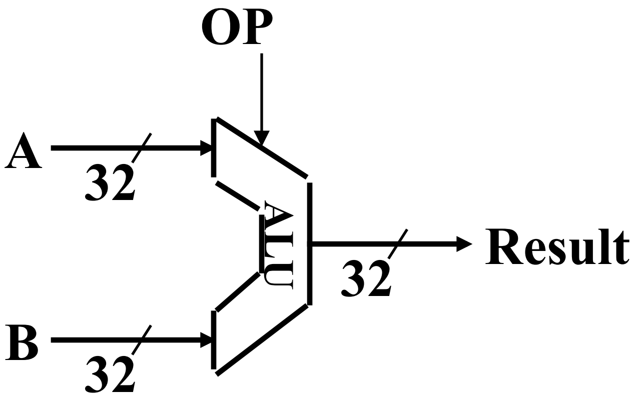
- Adder



- MUX



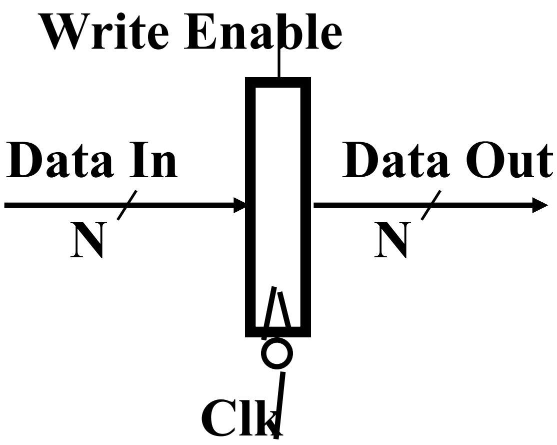
- ALU



Storage Element: Register (Basic Building Block)

° Register

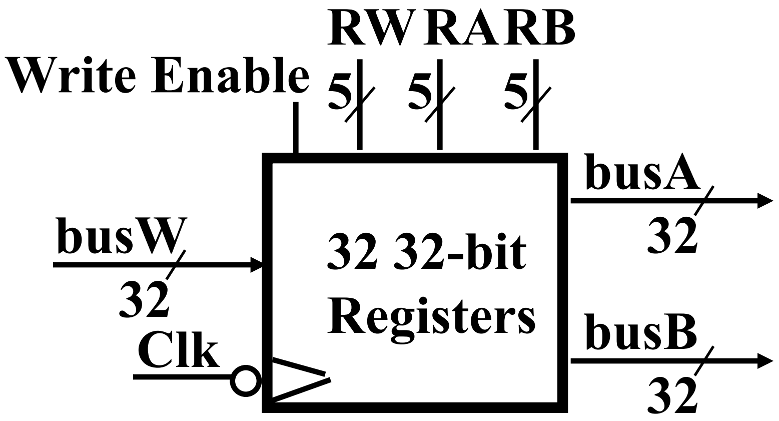
- Similar to the D Flip Flop except
  - N-bit input and output
  - Write Enable input
- Write Enable:
  - negated (0): Data Out will not change
  - asserted (1): Data Out will become Data In



Storage Element: Register File

- Register File consists of 32 registers:
  - Two 32-bit output busses:

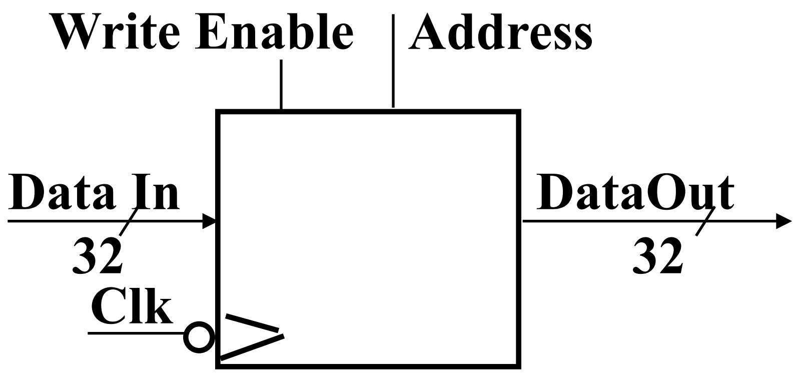
    busA and busB
  - One 32-bit input bus: busW



- Register is selected by:
  - RA (number) selects the register to put on busA (data)
  - RB (number) selects the register to put on busB (data)
  - RW (number) selects the register to be written via busW (data) when Write Enable is 1

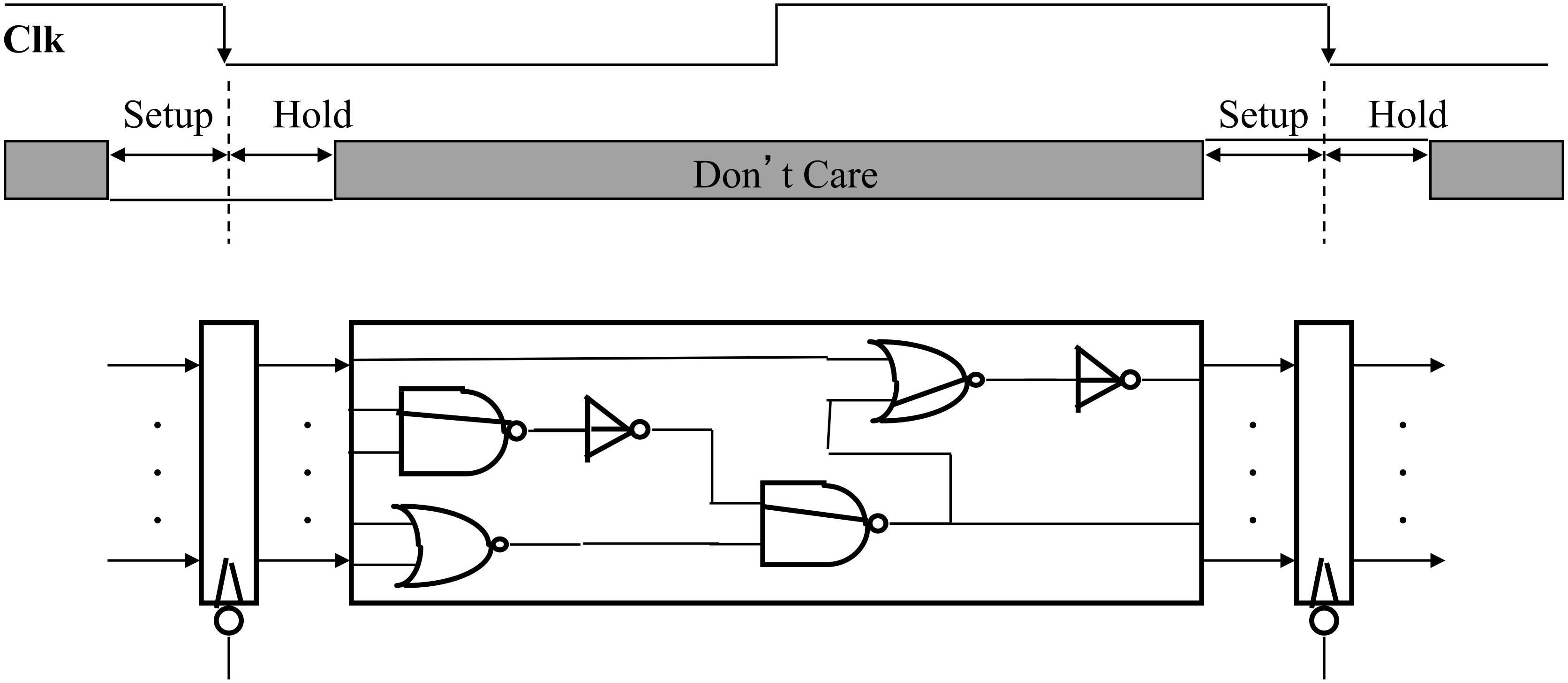
- Clock input (CLK)
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - RA or RB valid => busA or busB valid after “access time.”

Storage Element: Idealized Memory



- Memory (idealized)
  - One input bus: Data In
  - One output bus: Data Out
- Memory word is selected by:
  - Address selects the word to put on Data Out
  - Write Enable = 1: address selects the memory word to be written via the Data In bus
- Clock input (CLK)
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - Address valid => Data Out valid after “access time.”

Clocking Methodology



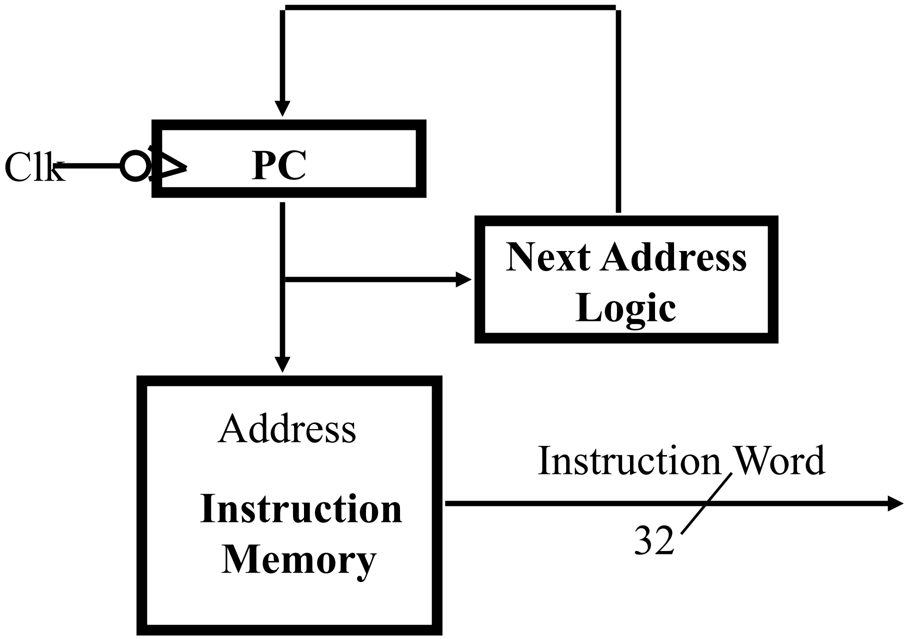
- All storage elements are clocked by the same clock edge
- Cycle Time = CLK-to-Q + Longest Delay Path + Setup + Clock Skew

Step 3

- Register Transfer <u>Requirements</u>
  –> Datapath <u>Assembly</u>
- Instruction Fetch
- Read Operands and Execute Operation

3a: Overview of the Instruction Fetch Unit

- The common RTL operations
  - Fetch the Instruction: mem[PC]
  - Update the program counter:
    - Sequential Code: PC <- PC + 4
    - Branch and Jump: PC <- “something else”



RTL: The ADD Instruction

31 26 21 16 11 6 0

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

° **add rd, rs, rt**

- **mem[PC]** **Fetch the instruction from memory**
- **R[rd] <- R[rs] + R[rt]** **The actual operation**
- **PC <- PC + 4** **Calculate the next instruction's address**

RTL: The Subtract Instruction

31 26 21 16 11 6 0

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

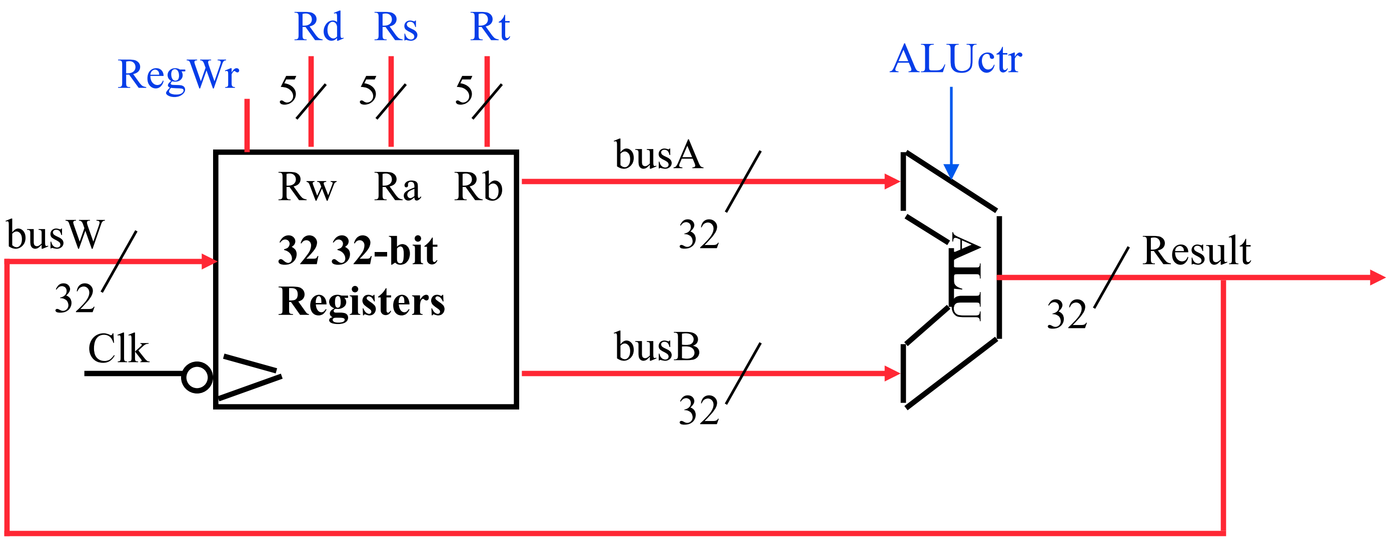
- sub rd, rs, rt
  - mem[PC] Fetch the instruction from memory
  - R[rd] <- R[rs] - R[rt] The actual operation
  - PC <- PC + 4 Calculate the next instruction's address

3b: Add & Subtract

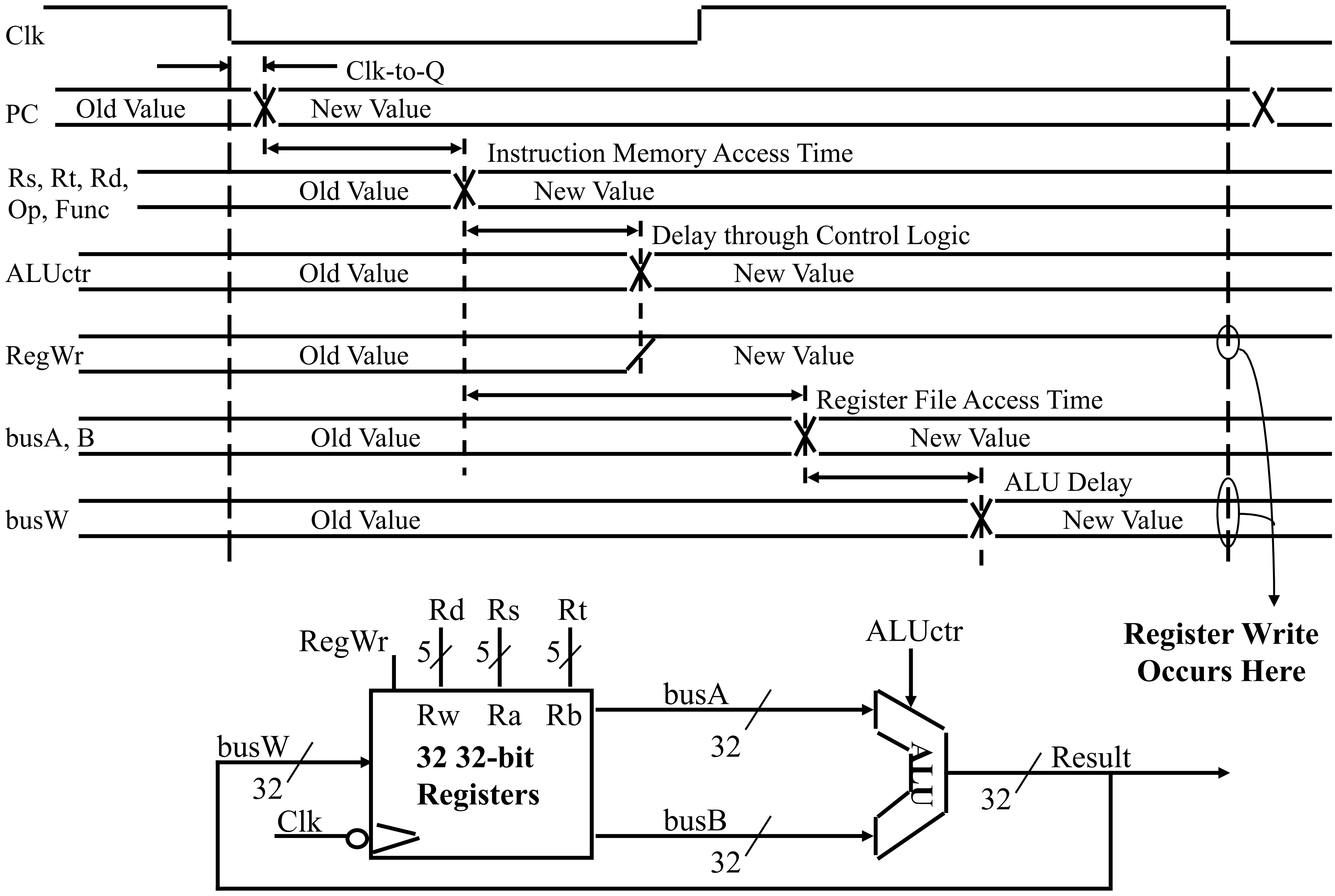
- R[rd] <- R[rs] op R[rt]   Example: addU rd, rs, rt
  - Ra, Rb, and Rw come from instruction’s rs, rt, and rd fields
  - ALUctr and RegWr: control logic after decoding the instruction

31 26 21 16 11 6 0

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits



Register-Register Timing



Register Write Occurs Here

RTL: The OR Immediate Instruction

31 26	21	16	0
op	**rs**	**rt**	**immediate**
6 bits	5 bits	5 bits	16 bits

° **ori rt, rs, imm16**

- **mem[PC]** **Fetch the instruction from memory**
- **R[rt] <- R[rs] or ZeroExt(imm16)**

  The OR operation
- **PC <- PC + 4** **Calculate the next instruction's address**

31 16	15 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	**immediate**
16 bits	16 bits

3c: Logical Operations with Immediate

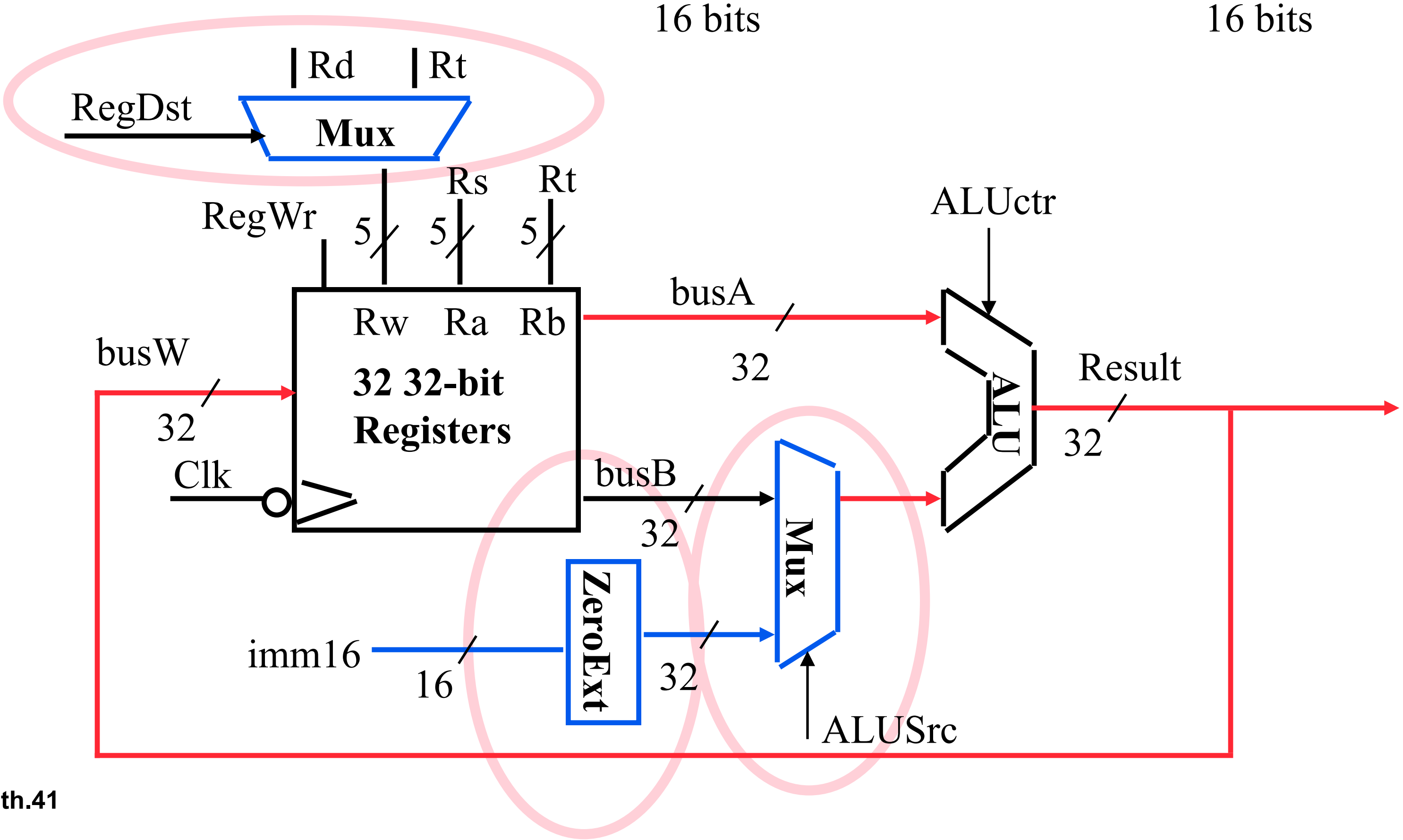
- R[rt] <- R[rs] op ZeroExt[imm16]

31 26 21 16 11 0

op	rs	rt	immediate
6 bits	5 bits	5 bits	rd? 16 bits

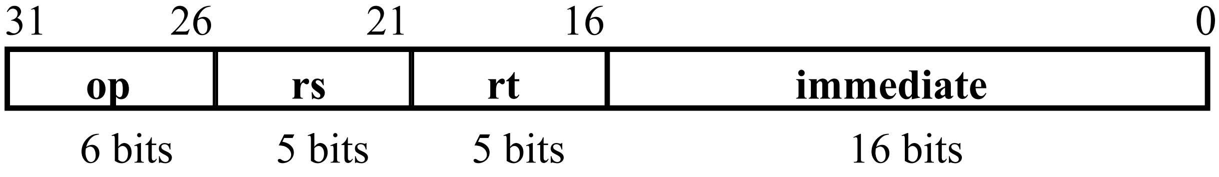
31 16 15 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	immediate
16 bits	16 bits



RTL: The Load Instruction

° **lw rt, rs, imm16**



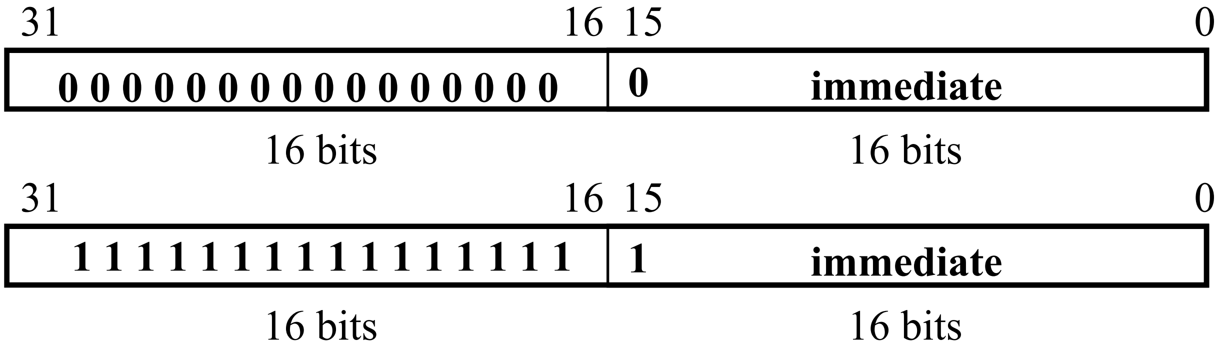
31 26	21	16	0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

- **mem[PC]** **Fetch the instruction from memory**
- **Addr <- R[rs] + SignExt(imm16)**

  Calculate the memory address

  R[rt] <- Mem[Addr] **Load the data into the register**

- **PC <- PC + 4** **Calculate the next instruction's address**



31 16	15 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 immediate
16 bits	16 bits

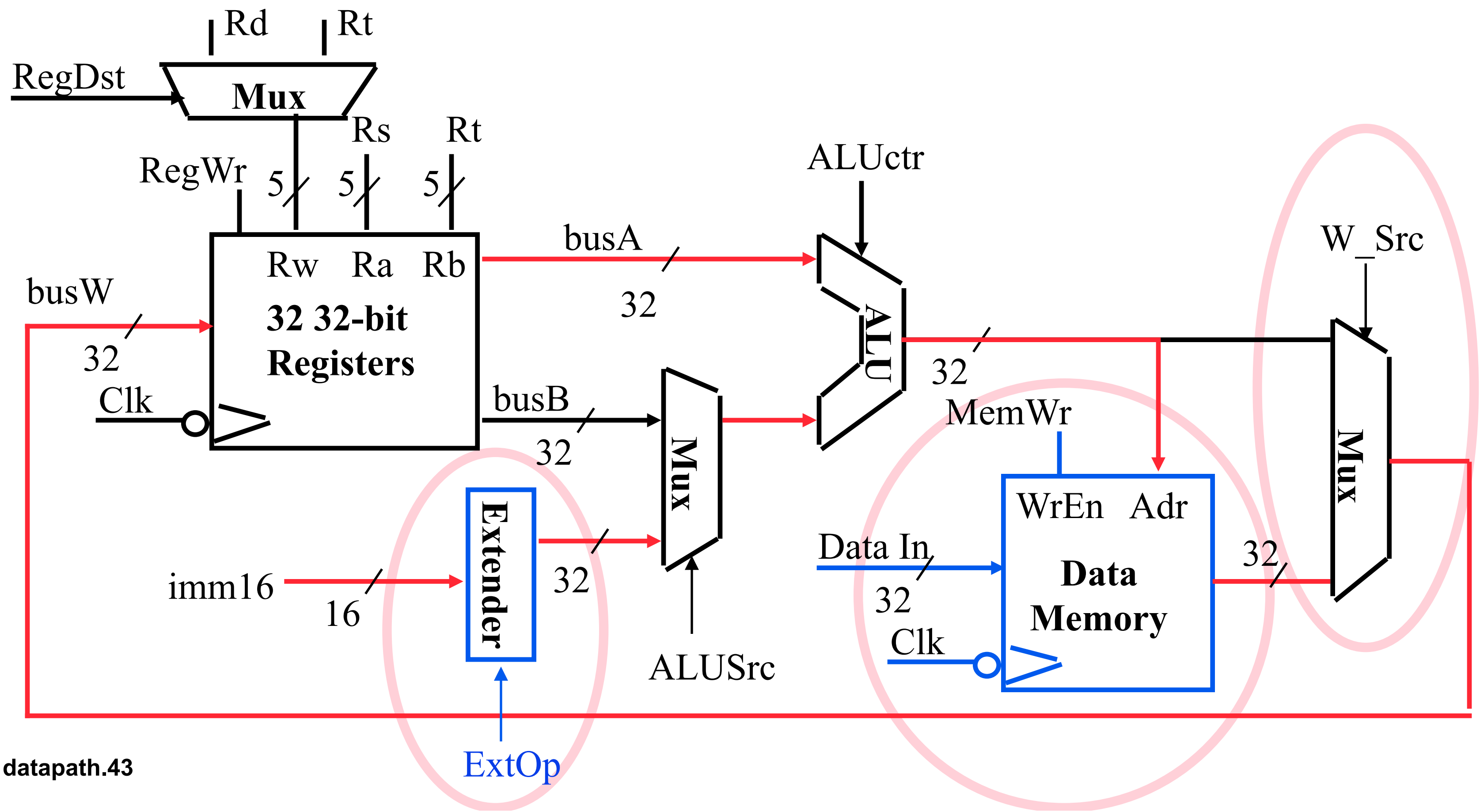
31 16	15 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	1 immediate
16 bits	16 bits

3d: Load Operations

- R[rt] <- Mem[R[rs] + SignExt[imm16]] Example: lw rt, rs, imm16

31 26 21 16 11 0

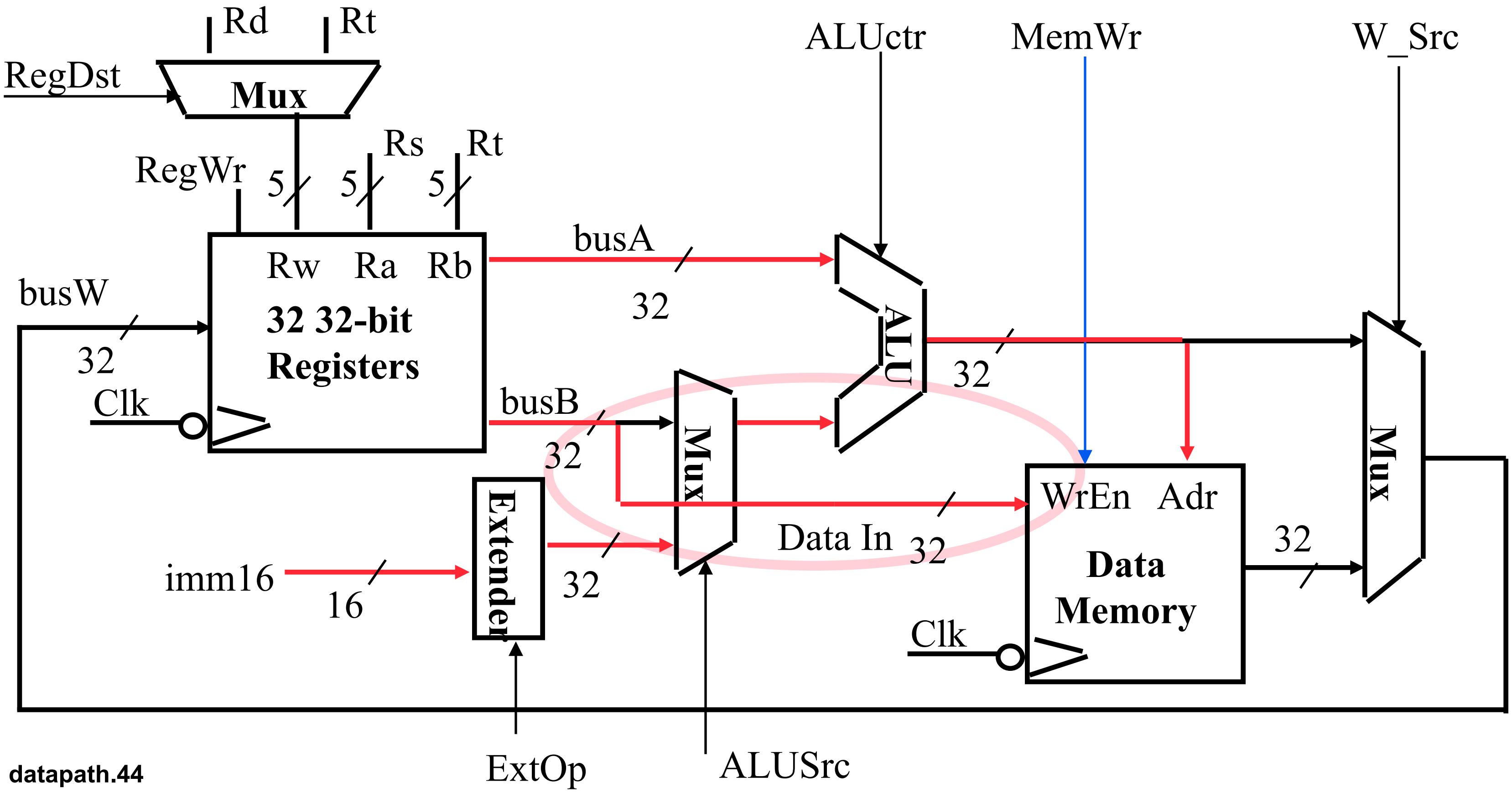
op	rs	rt	immediate
6 bits	5 bits	5 bits	rd 16 bits



3e: Store Operations

- Mem[R[rs] + SignExt[imm16]] <- R[rt]   Example: sw   rt, rs, imm16

31 26	21	16	0
op	**rs**	**rt**	**immediate**
6 bits	5 bits	5 bits	16 bits



Summary

- **5 steps to design a processor**
  - 1. Analyze instruction set => datapath requirements
  - 2. Select set of datapath components & establish clock methodology
  - 3. Assemble datapath meeting the requirements
  - 4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
  - 5. Assemble the control logic
- **MIPS makes it easier**
  - Instructions same size
  - Source registers always in same place
  - Immediates same size, location
  - Operations always on registers/immediates
- **Single cycle datapath**
  => CPI=1, CCT (clock cycle time) = long
- **Next time: other insts and implementing control**