Chap. 3

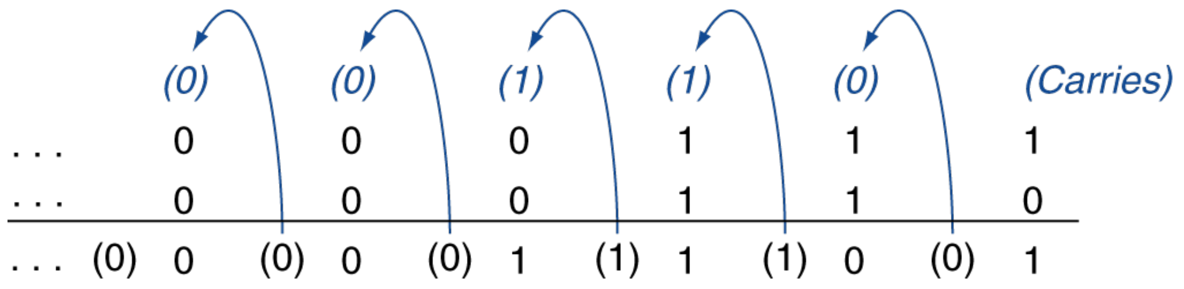
Arithmetic for Computers

Arithmetic for Computers

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- Floating-point (浮動小數點) real numbers
  - Representation and operations

Integer Addition

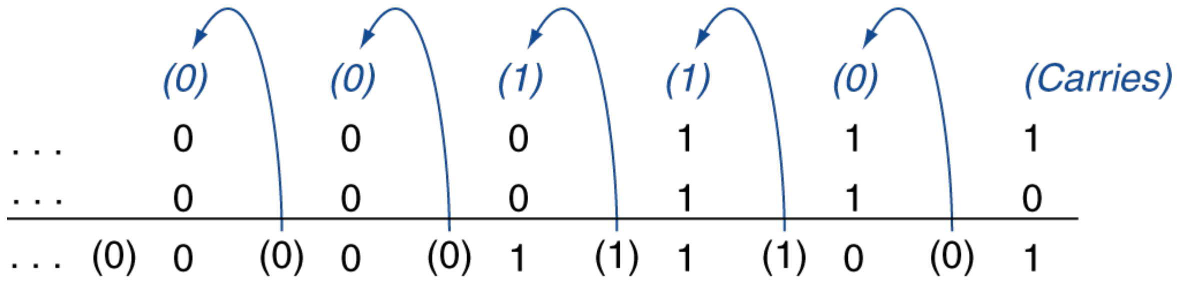
- Example : 7 + 6



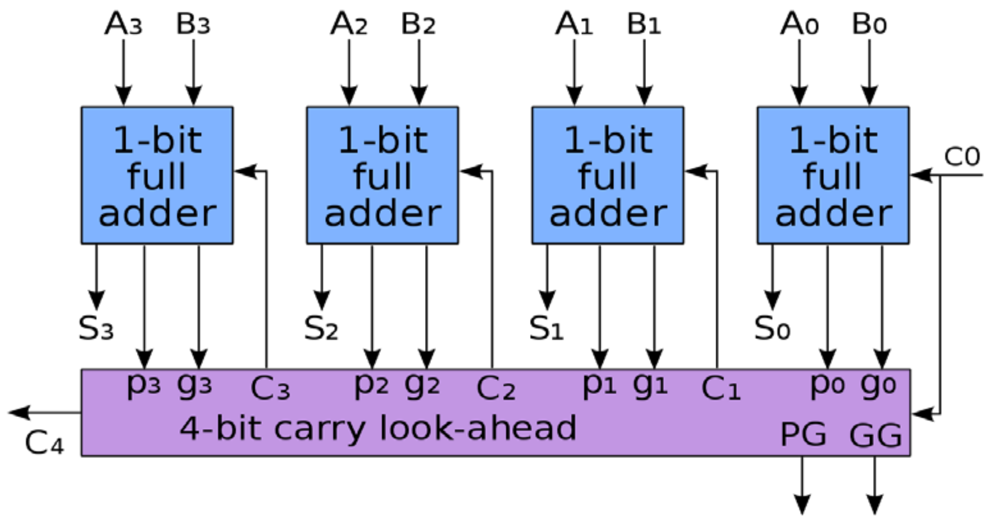
- Half Adder
- Full Adder
- Parallel Adder
- Carry Lookahead Adder

Integer Addition

- Example : 7 + 6



carry lookahead adder(CLA)

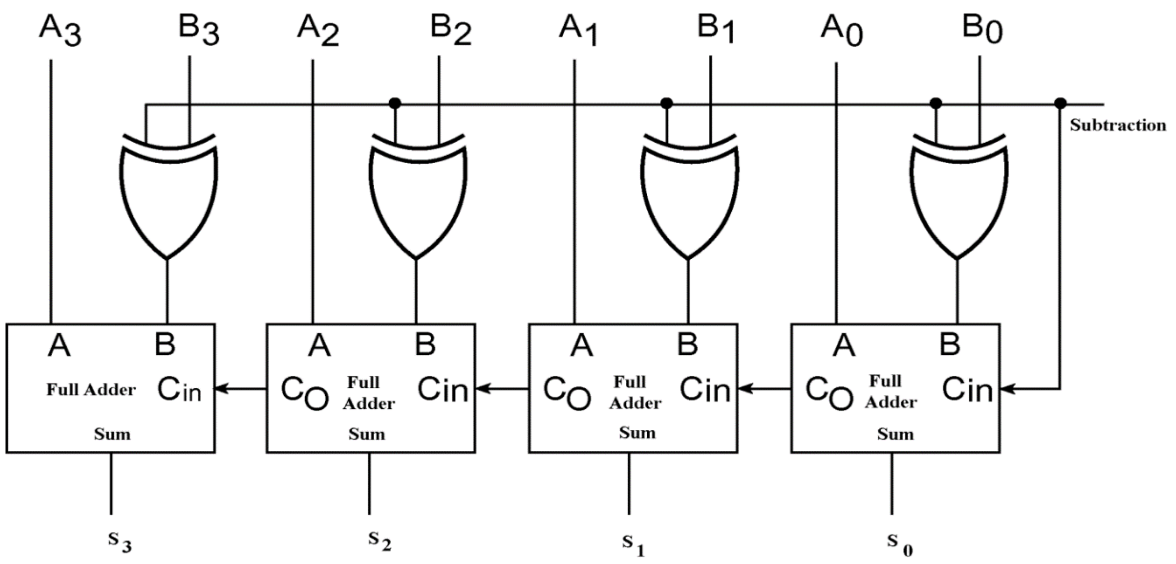


Integer Subtraction

- Add negation of second operand
- Example : 7 – 6 = 7 + (–6)

```
+7:        0000 0000 … 0000 0111
–6:        1111 1111 … 1111 1010
--------------------------------
+1:        0000 0000 … 0000 0001
```

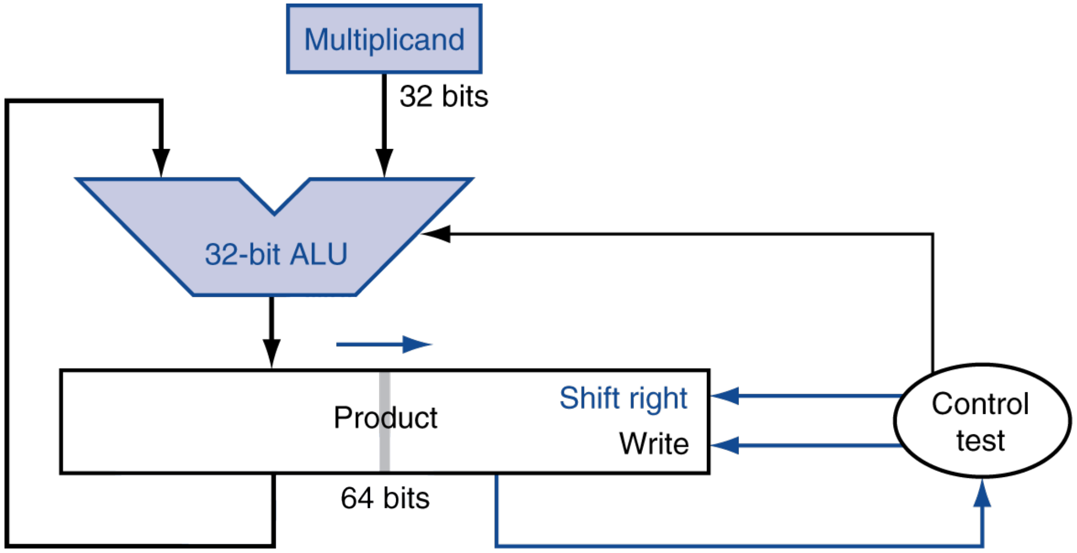
- Binary Adder/Subtractor



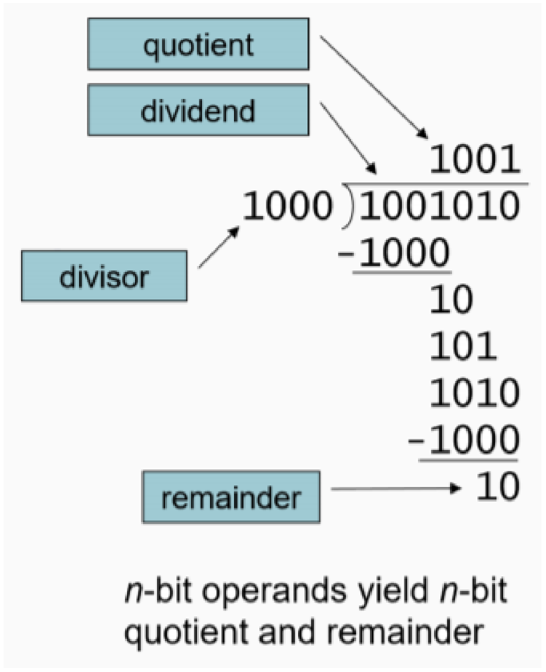
Optimized Multiplier (Fig.3.5)

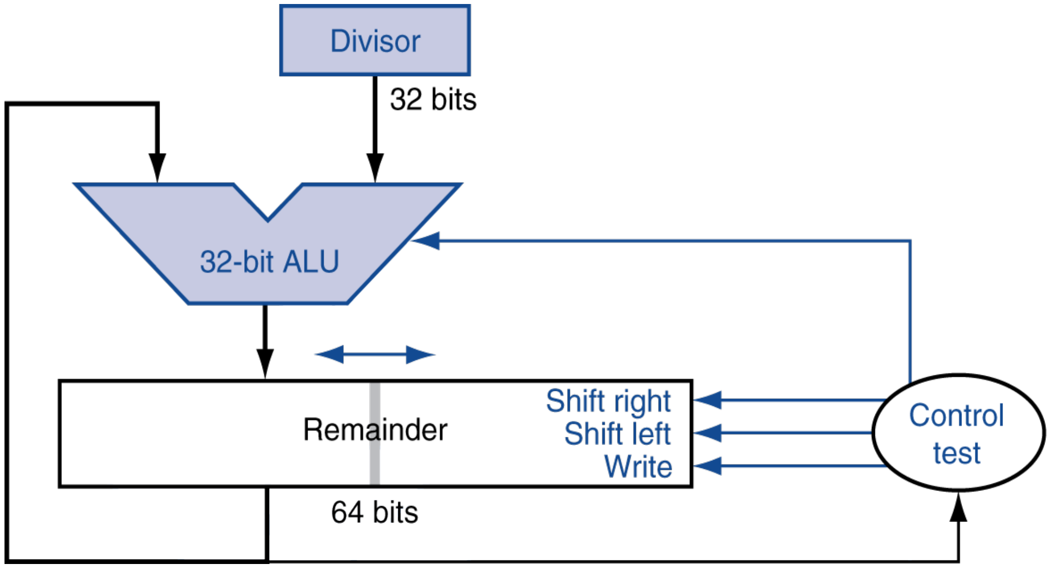
- Perform steps in parallel: add/shift

```
      1000
  x   1001
  =======
      1000
     0000
    0000
  +1000
 ========
  1001000
```



Optimized Divider (Fig. 3.11)





- Looks a lot like a multiplier!
  - Same hardware can be used for both

Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56} ← normalized
  - $+0.002 \times 10^{-4}$ ← not normalized
  - $+987.02 \times 10^{9}$ ← not normalized
- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types `float` and `double` in C

Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted

- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)

  - Precision vs. range

IEEE Floating-Point Format

single: 8 bits
double: 11 bits

single: 23 bits
double: 52 bits

S	Exponent	Fraction

$$x = (-1)^{S} \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: sign bit (0 $\Rightarrow$ non-negative, 1 $\Rightarrow$ negative)
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203

Subword Parallellism

- Graphics and audio applications can take advantage of performing simultaneous operations on short vectors
  - Example: 128-bit adder:
    - Sixteen 8-bit adds
    - Eight 16-bit adds
    - Four 32-bit adds
- Also called data-level parallelism, vector parallelism, or
- Single Instruction, Multiple Data (SIMD)

x86 FP Architecture

- Originally based on **8087 FP coprocessor**
  - 8 × 80-bit extended-precision registers
  - Used as a push-down stack
  - Registers indexed from TOS: ST(0), ST(1), …
- Very difficult to generate and optimize code
  - Result : poor FP performance

Streaming SIMD Extension 2 (SSE2)

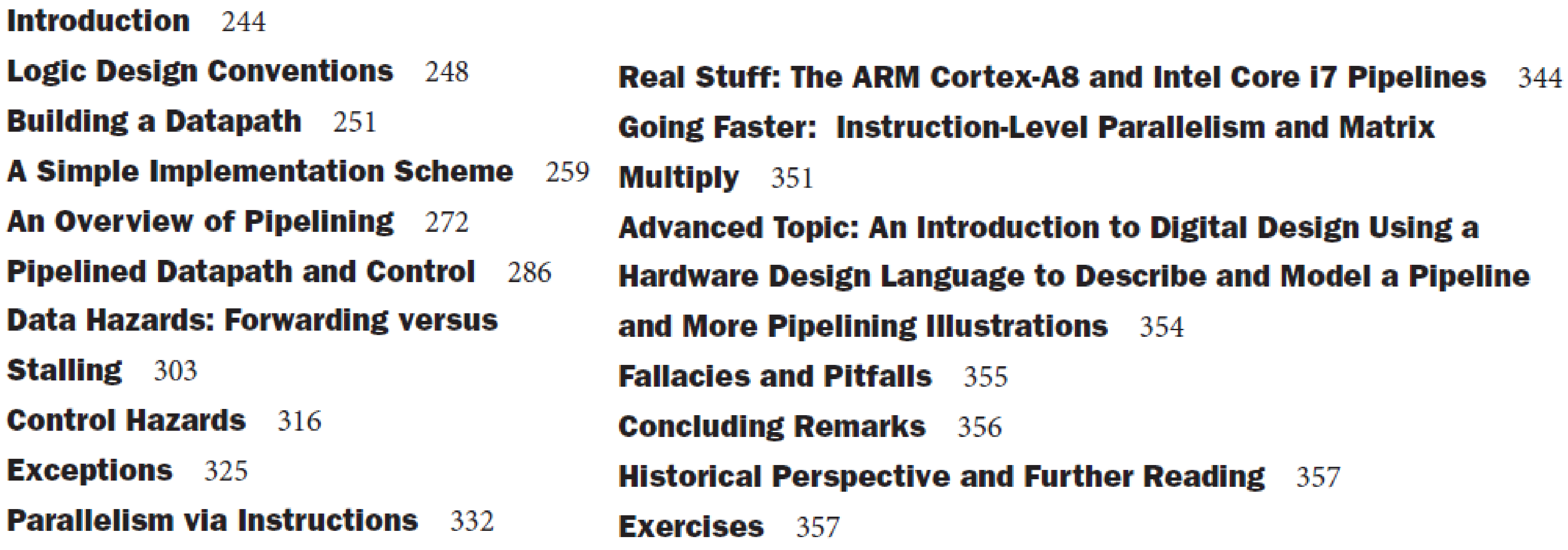
- Can be used for multiple FP operands
  - 2 × 64-bit double precision
  - 4 × 32-bit double precision
  - <u>S</u>ingle-<u>I</u>nstruction <u>M</u>ultiple-<u>D</u>ata

Concluding Remarks

- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow
- MIPS ISA
  - Core instructions : 54 most frequently used
    - 100% of SPECINT, 97% of SPECFP
  - Other instructions: less frequent

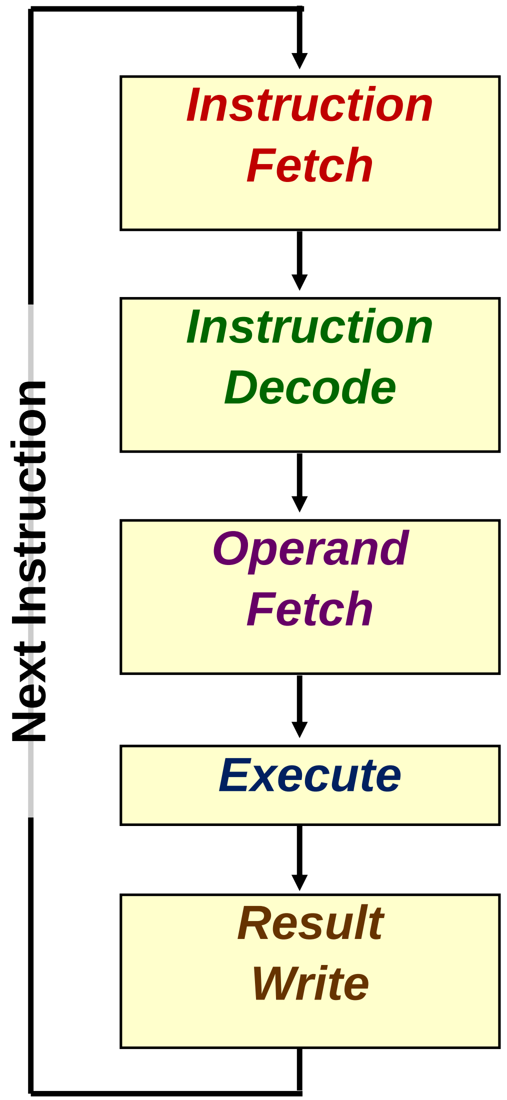
Chap. 4 The Processor

Part A – Simple Implementation



** NUS, Aaron Tan 교수의 강의자료를 일부 포함하고 있습니다. **

Instruction Execution Cycle (Basic)



Fetch:

- Get instruction from memory
- Address is in **P**rogram **C**ounter (PC) Register

Decode:

- Find out the operation required

Operand Fetch:

- Get operand(s) needed for operation

Execute:

- Perform the required operation

Result Write (Store):

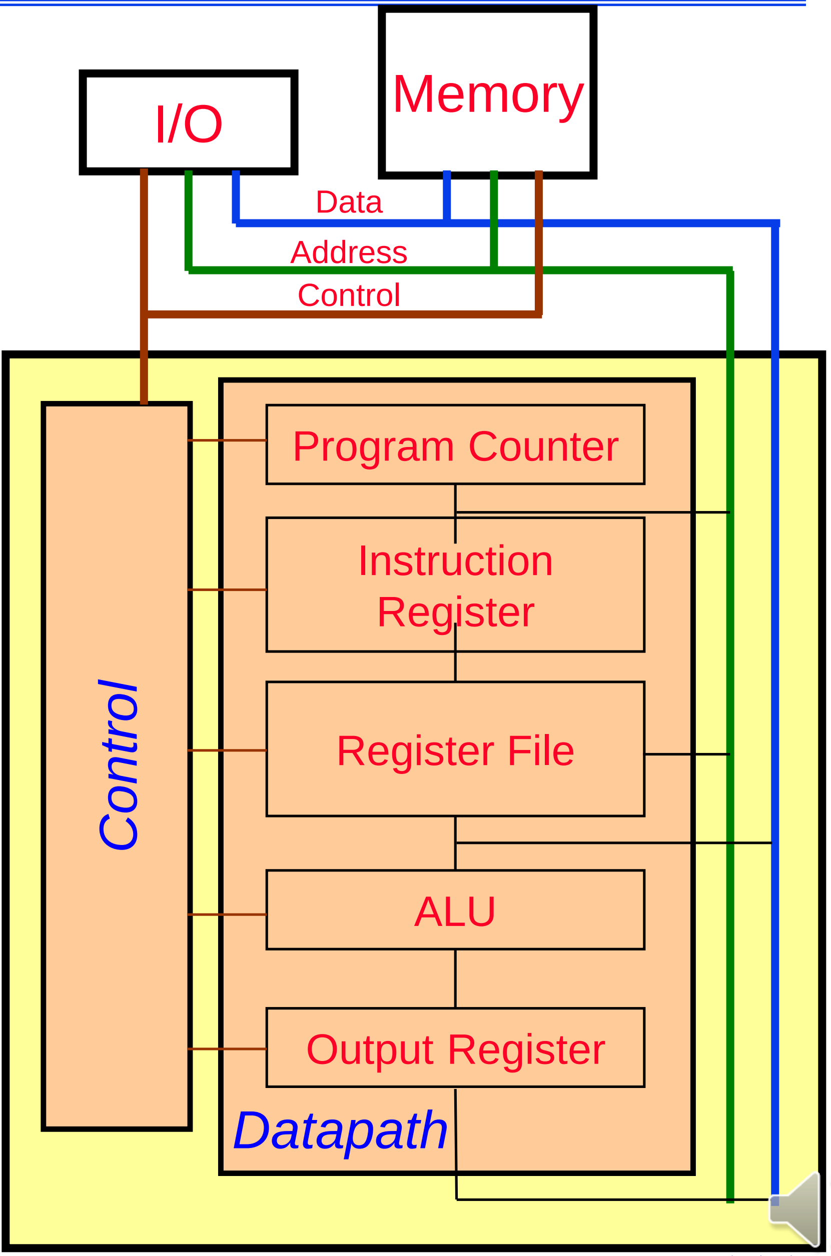
- Store the result of the operation

The Processor

- 2 Major Components for a processor
  - Datapath
    - Performs the arithmetic, logical and memory operations
    - Collection of components that process data

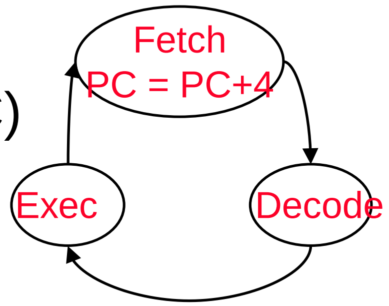
      Arithmetic Logic Unit(ALU),

      Shifters, Registers, ultipliers
  - Control
    - Tells the datapath, memory and I/O devices what to do according to program instructions



The Processor (Datapath & Control) Implementation

- Our implementation of the MIPS is simplified
  - arithmetic and logical instructions: `add, sub, and, or, slt`
  - memory-reference instructions: `lw, sw`
  - control flow instructions: `beq, bne, j`
- Generic implementation
  - use the PC to supply the instruction address and fetch the instruction from memory (and update the PC)
  - decode the instruction (and read registers)
  - execute the instruction



- All instructions (except `j`) use the ALU after reading the registers
  - *Arithmetic result : add, addi, sub, and, or*
  - *Memory address for load/store*
  - *Comparison result for branches*

Clocking Methodologies

- The clocking methodology defines when data in a state element is valid and stable relative to the clock
  - State elements - a memory element such as a register
  - Edge triggered - all state changes occur on a clock edge

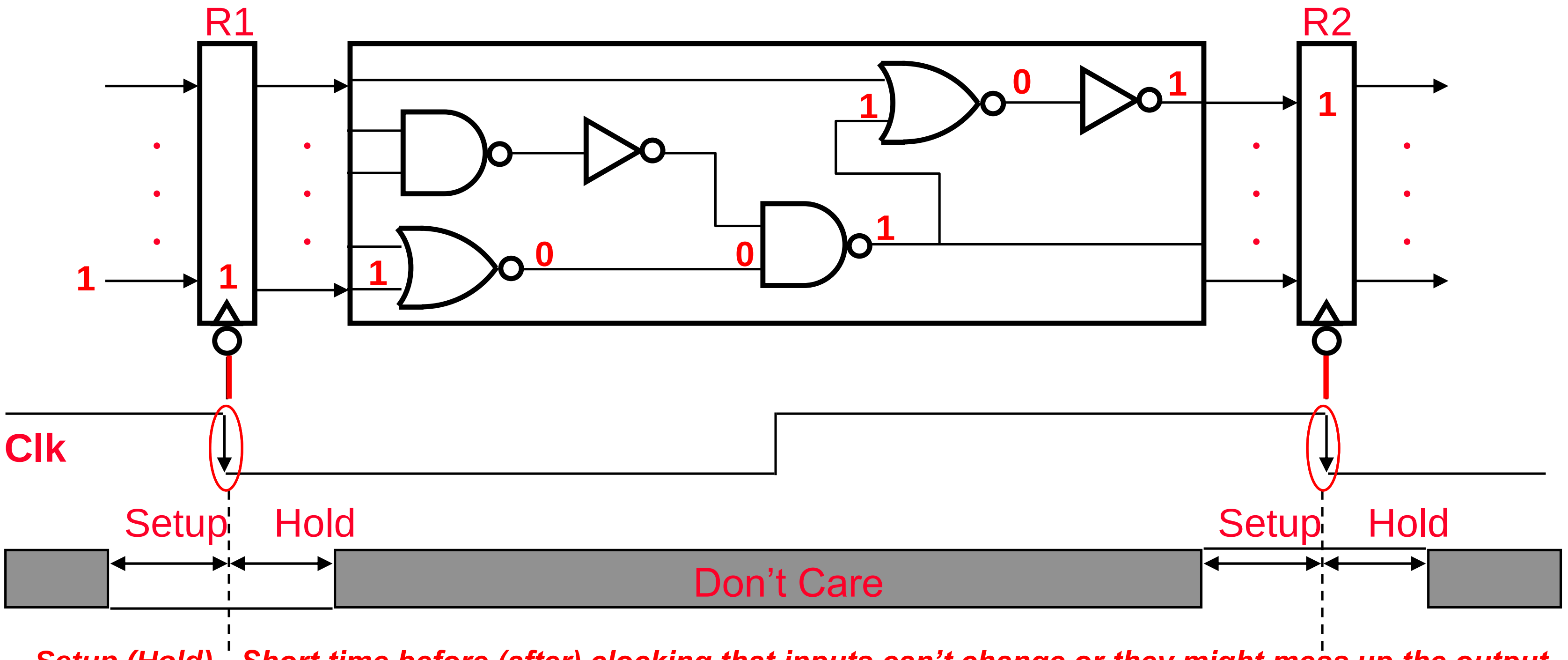
- Typical execution
  - read contents of state elements
  - send values through combinational logic
  - write results to one or more state elements

Register Transfer Language and Clocking

Register transfer in RTL:

R2 ← f(R1)

What Really Happens Physically



Setup (Hold) - Short time before (after) clocking that inputs can't change or they might mess up the output

Two possible clocking methodologies : positively triggered or negatively triggered. This class uses the negatively-triggered.

Building a Datapath

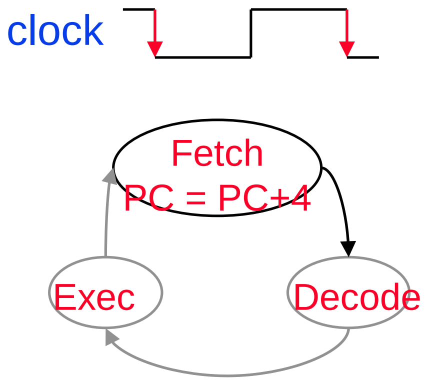
- MIPS Instruction Execution
- Design changes:
  - Merge *Decode* and *Operand Fetch* – Decode is simple for MIPS
  - Split *Execute* into ***ALU*** (Calculation) and ***Memory Access***

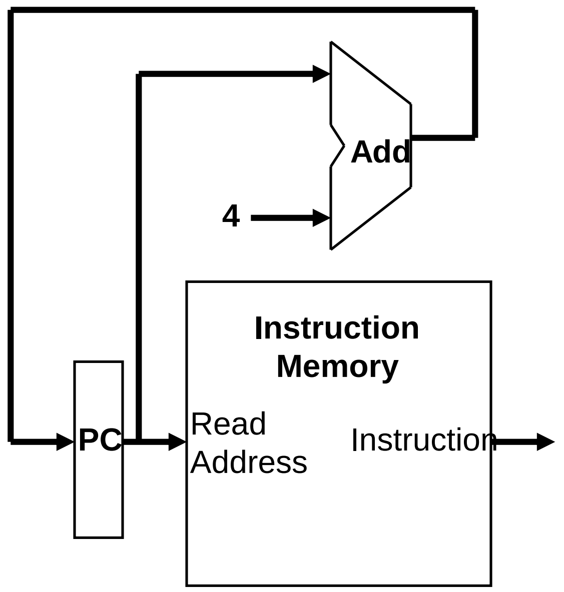
	`add $3, $1, $2`	`lw $3, 20( $1 )`	`beq $1, $2, label`
Fetch	Read inst. at [PC]	Read inst. at [PC]	Read inst. at [PC]
Decode & Operand Fetch	○ Read [**$1**] as *opr1* ○ Read [**$2**] as *opr2*	○ Read [**$1**] as *opr1* ○ Use ***20*** as *opr2*	○ Read [**$1**] as *opr1* ○ Read [**$2**] as *opr2*
ALU	*Result = opr1 + opr2*	*MemAddr = opr1 + opr2*	*Taken = (opr1 == opr2)?* *Target* = (**PC**+4) + **ofst**×**4**
Memory Access		Use *MemAddr* to read from memory	
Result Write	*Result* stored in **$3**	*Memory* data stored in **$3**	if (*Taken*) **PC** = *Target*

opr = operand **MemAddr** = Memory Address **ofst** = offset

Fetching Instructions

- Fetching instructions involves
  - reading the instruction from the Instruction Memory
  - updating the PC value to be the address of the next (sequential) instruction





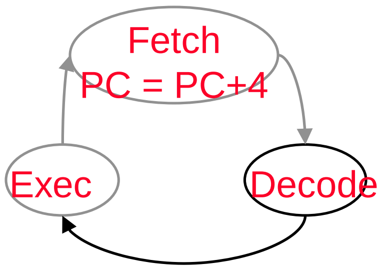
Memory

	………..
2048	`add $3, $1, $2`
2052	`sll $4, $3, 2`
2056	`andi $1, $4, 0xF`
……	………..

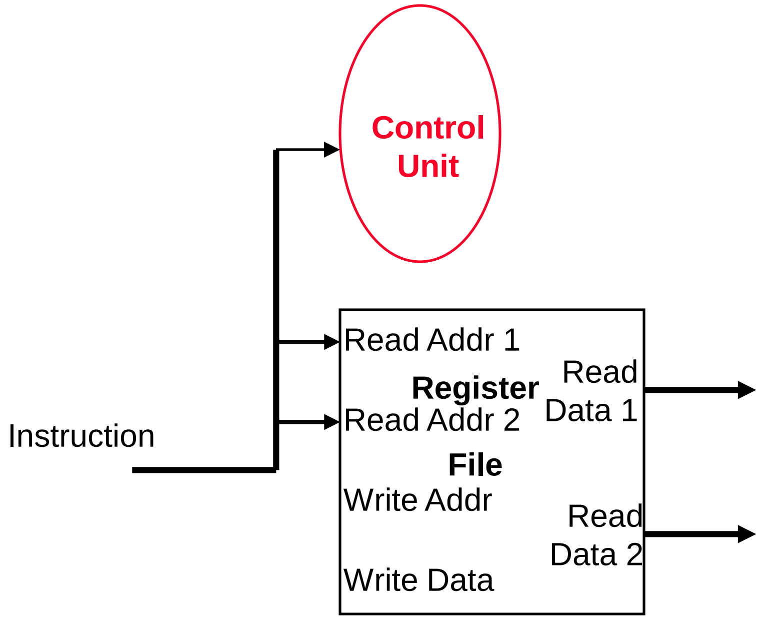
  - PC is updated every clock cycle, so it does not need an explicit write control signal just a clock signal
  - Reading from the Instruction Memory is a combinational activity, so it doesn't need an explicit read control signal

Decoding Instructions

- Decoding instructions involves
  - sending the fetched instruction's opcode and function field bits to the control unit

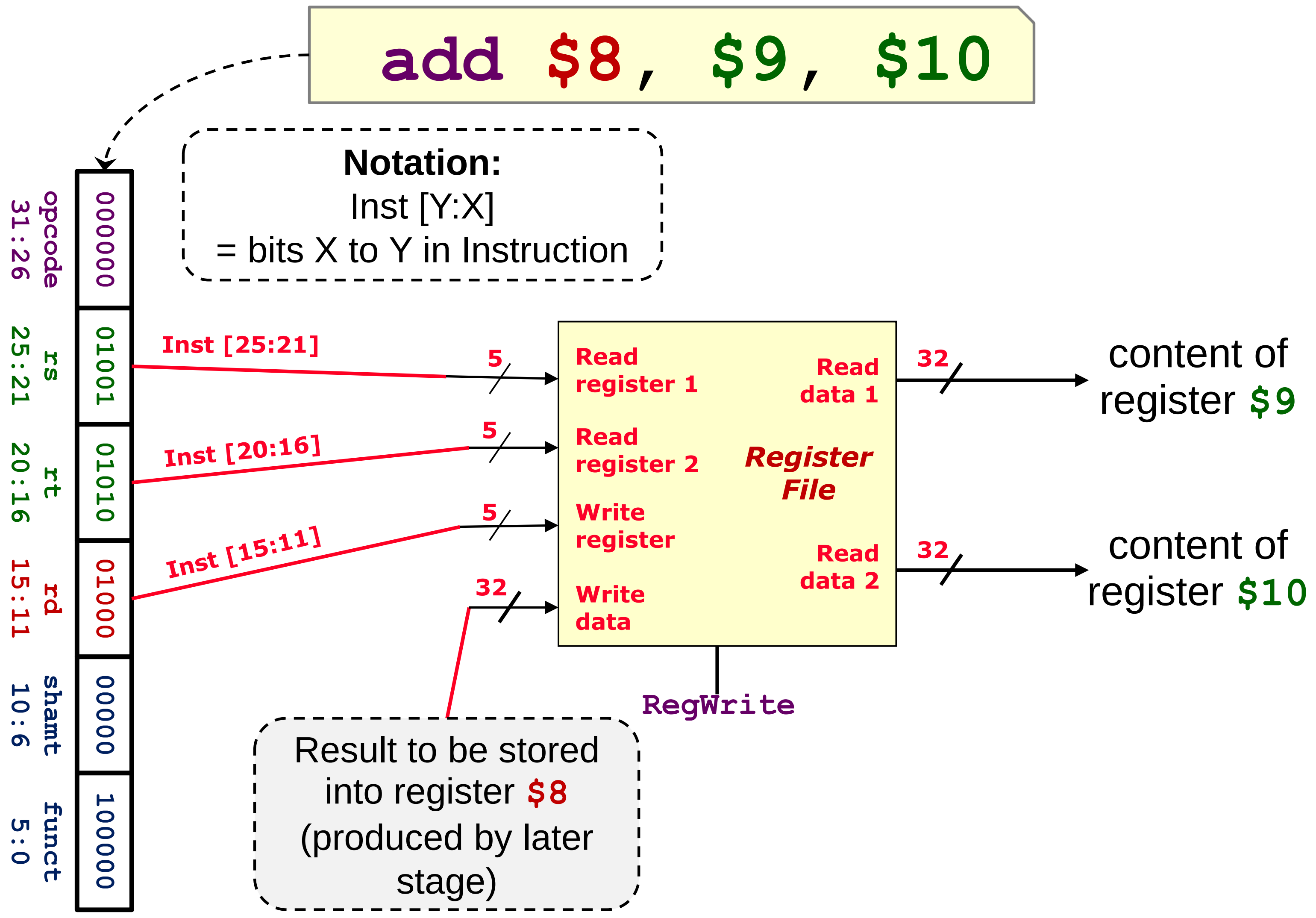


and

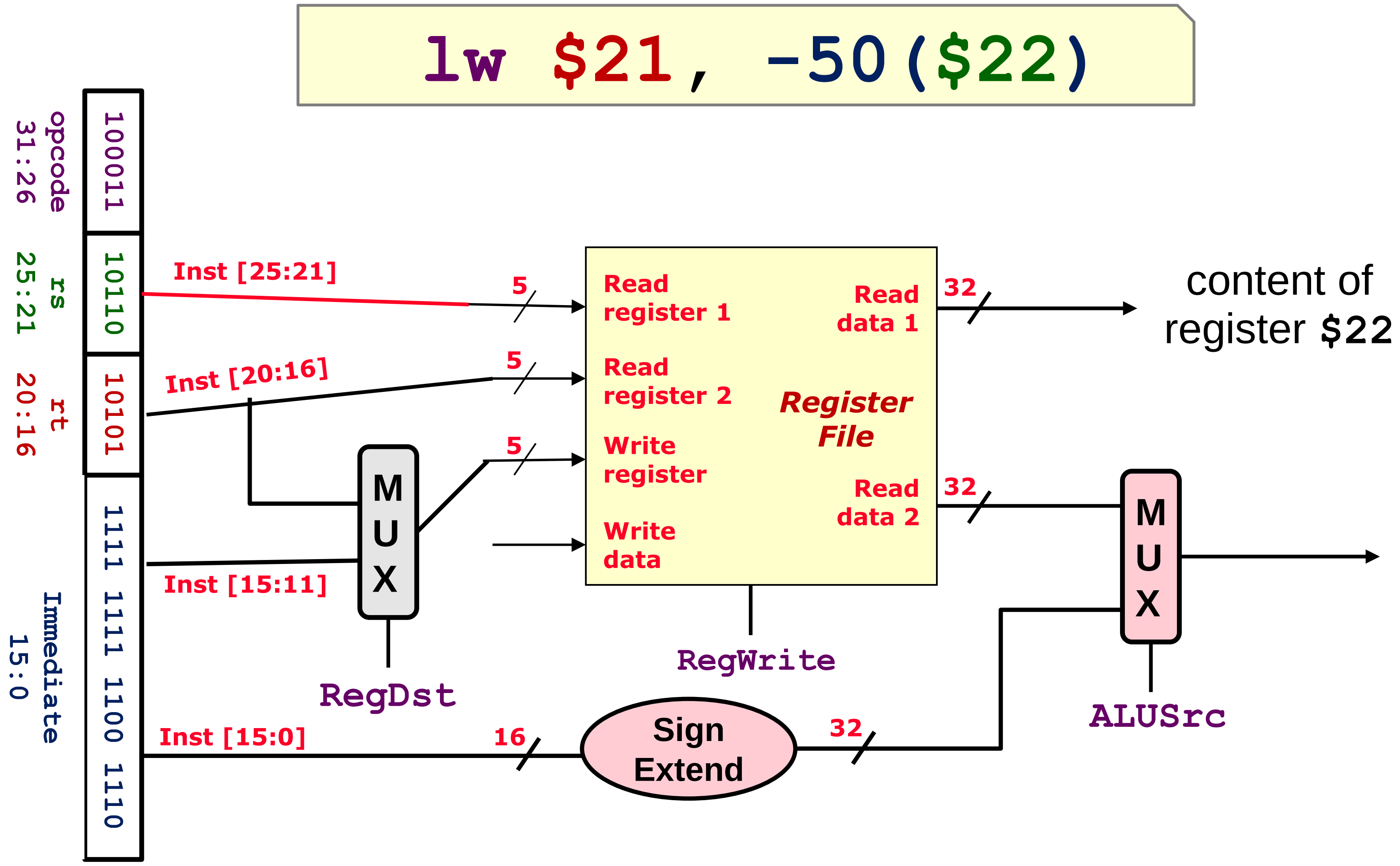


  - reading two values from the Register File
    - Register File addresses are contained in the instruction

Decode Stage: **R-Format Instruction**

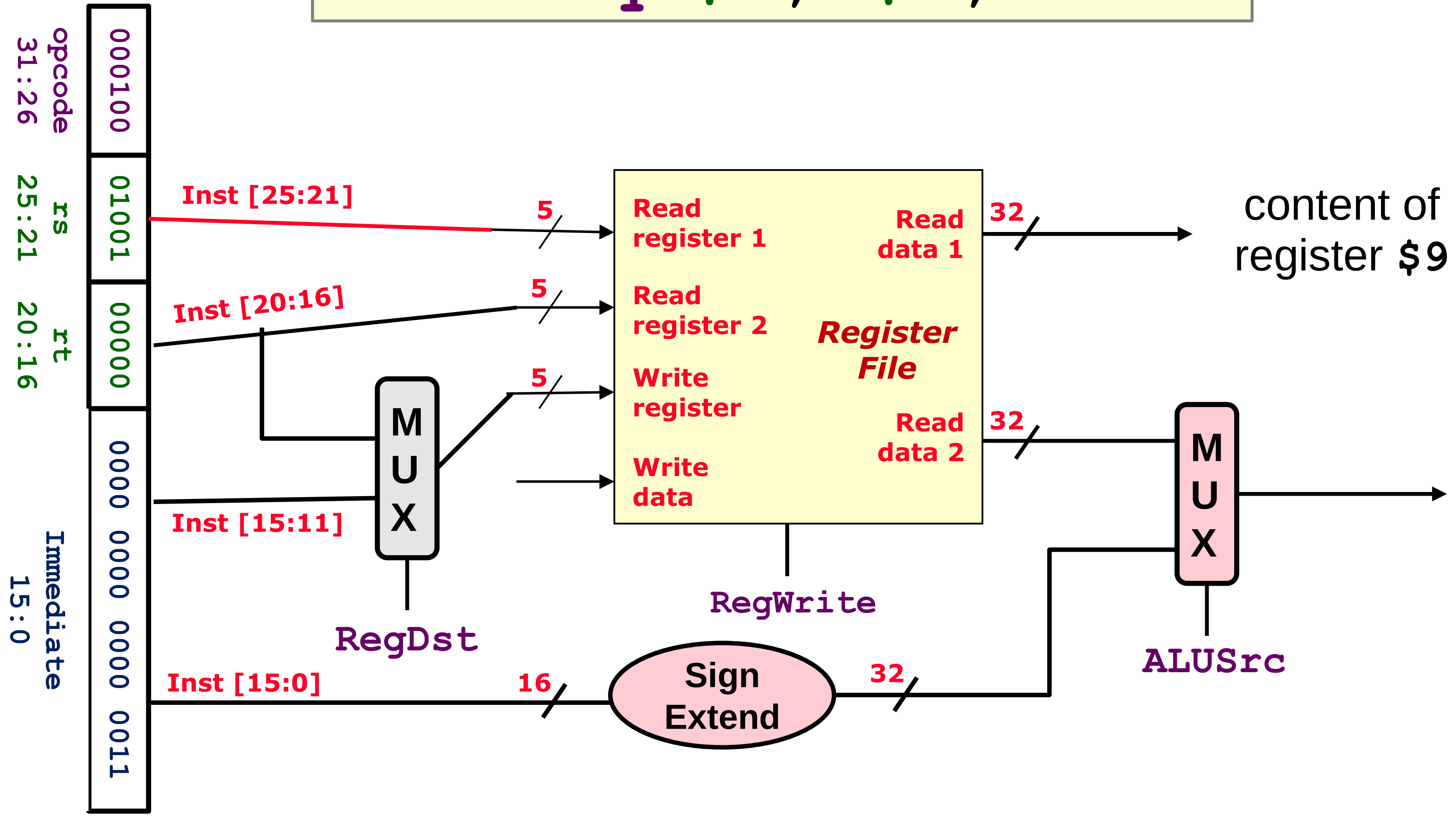


Decode Stage: **Load Word Instruction**



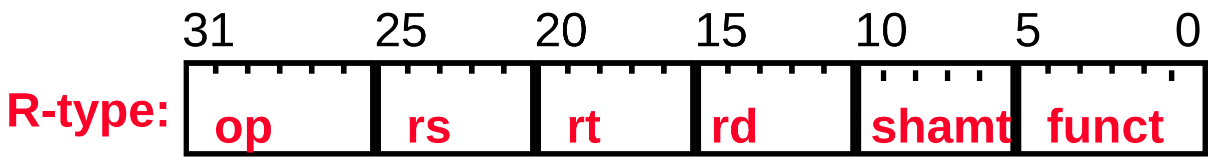
Decode Stage: **Branch Instruction**

`beq $9, $0, 3`

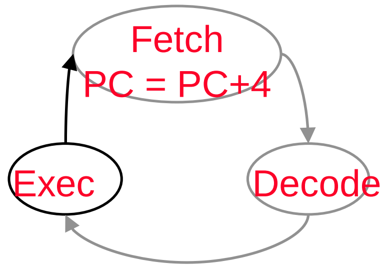


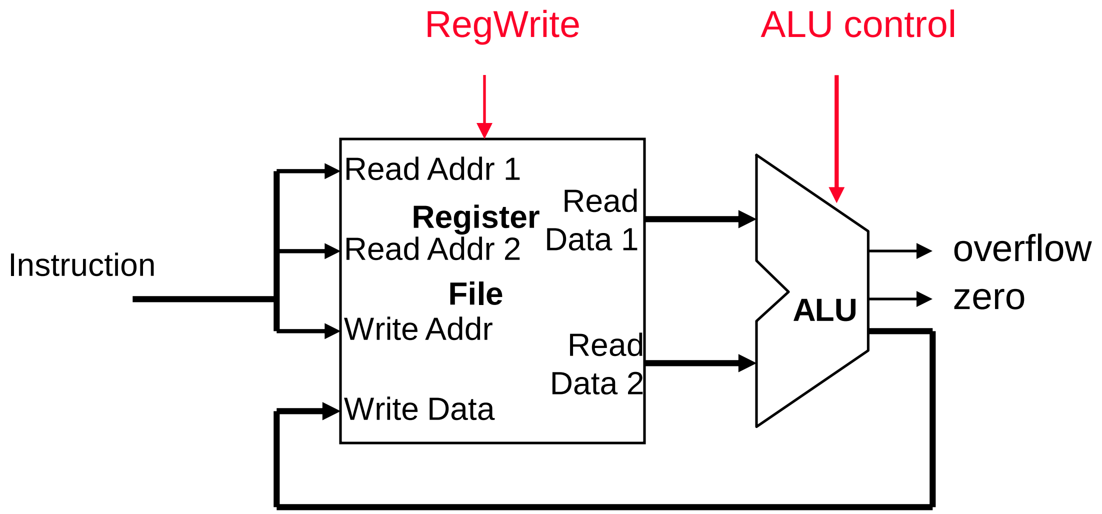
Executing R Format Operations

- R format operations (**add, sub, slt, and, or**)



- perform operation (op and funct) on values in rs and rt
- store the result back into the Register File (into location rd)

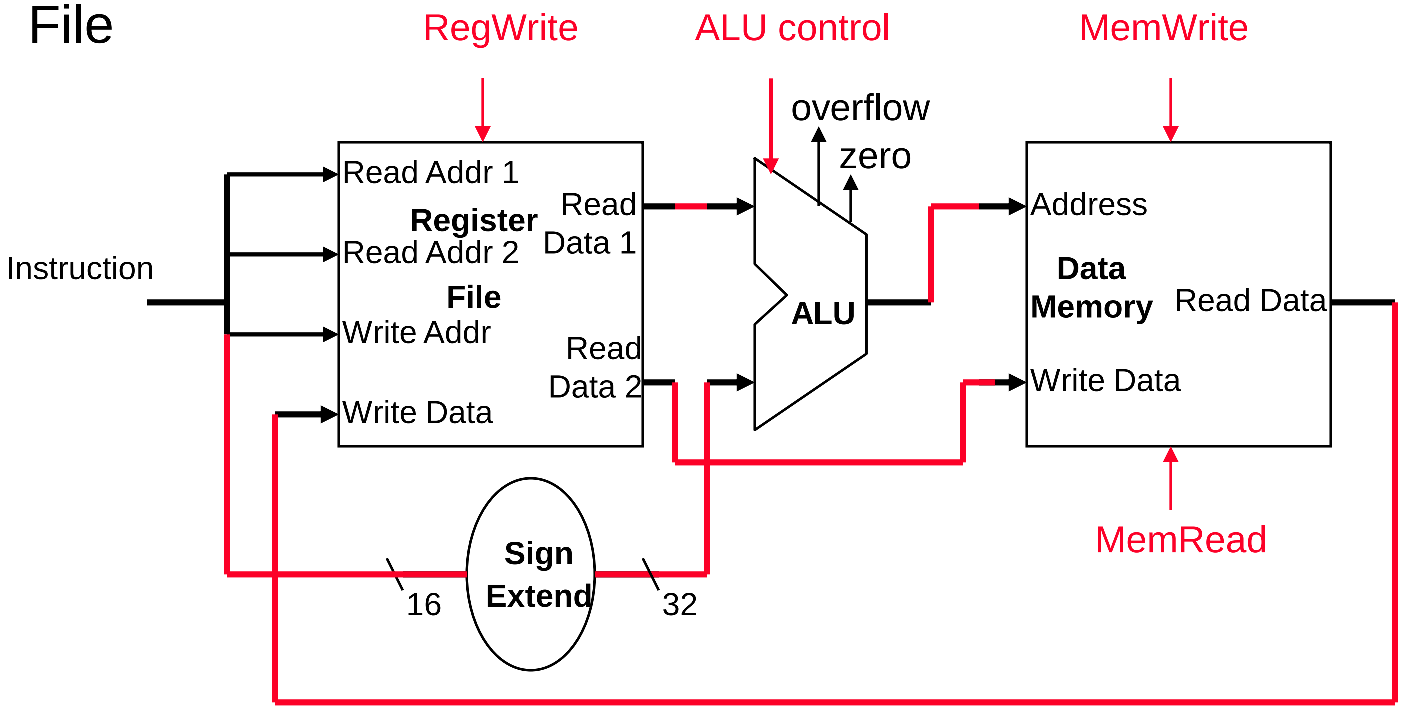




- Note that Register File is not written every cycle (e.g. **sw**), so we need an explicit write control signal for the Register File

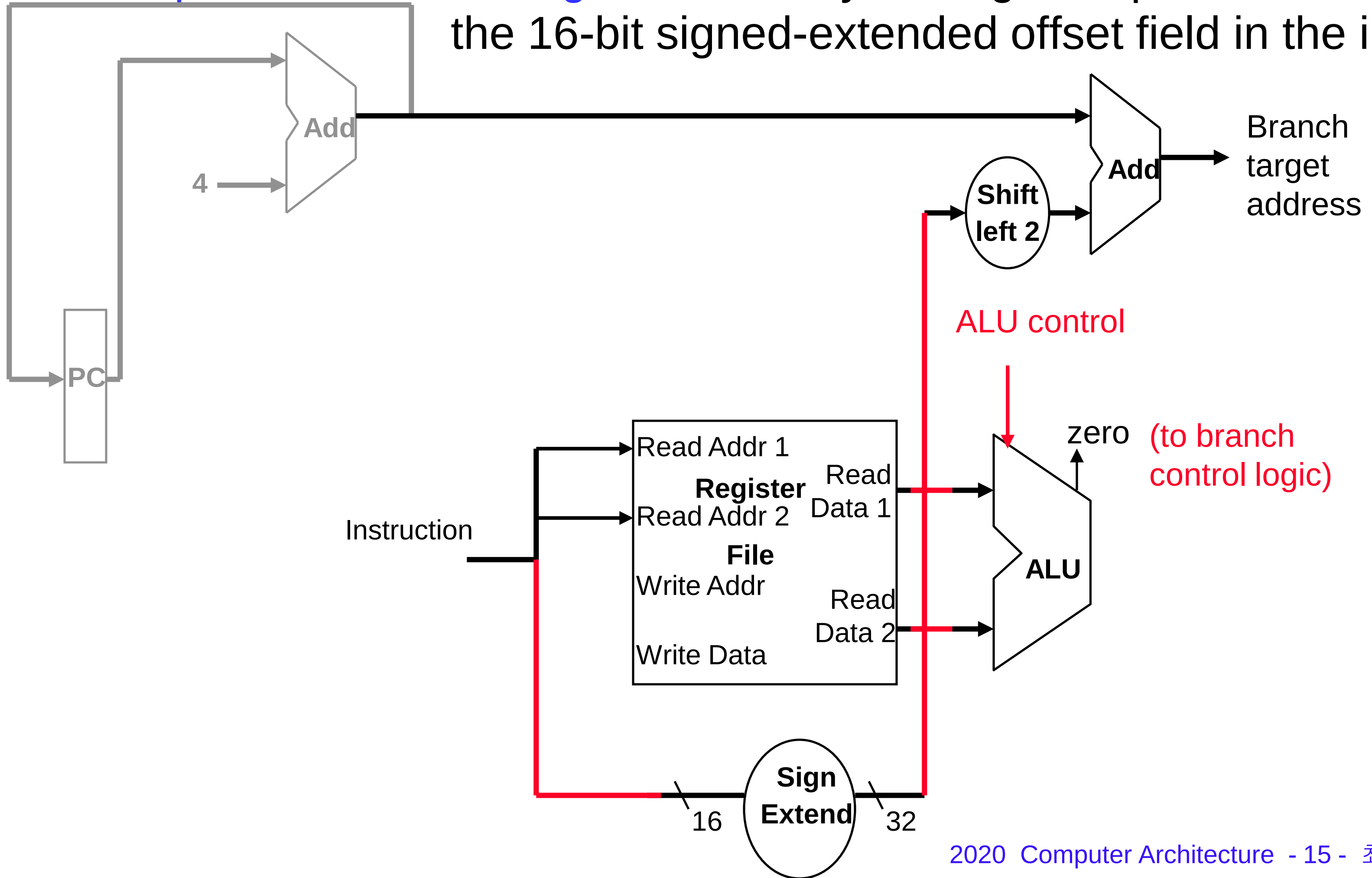
Executing Load and Store Operations

- Load and store operations involves
  - compute memory address by adding the base register (read from the Register File during decode) to the 16-bit signed-extended offset field in the instruction
  - store value (read from the Register File during decode) written to the Data Memory
  - load value, read from the Data Memory, written to the Register File



Executing Branch Operations

- Branch operations involves
  - compare the operands read from the Register File during decode for equality (`zero` ALU output)
  - compute the branch target address by adding the updated PC to the 16-bit signed-extended offset field in the instr



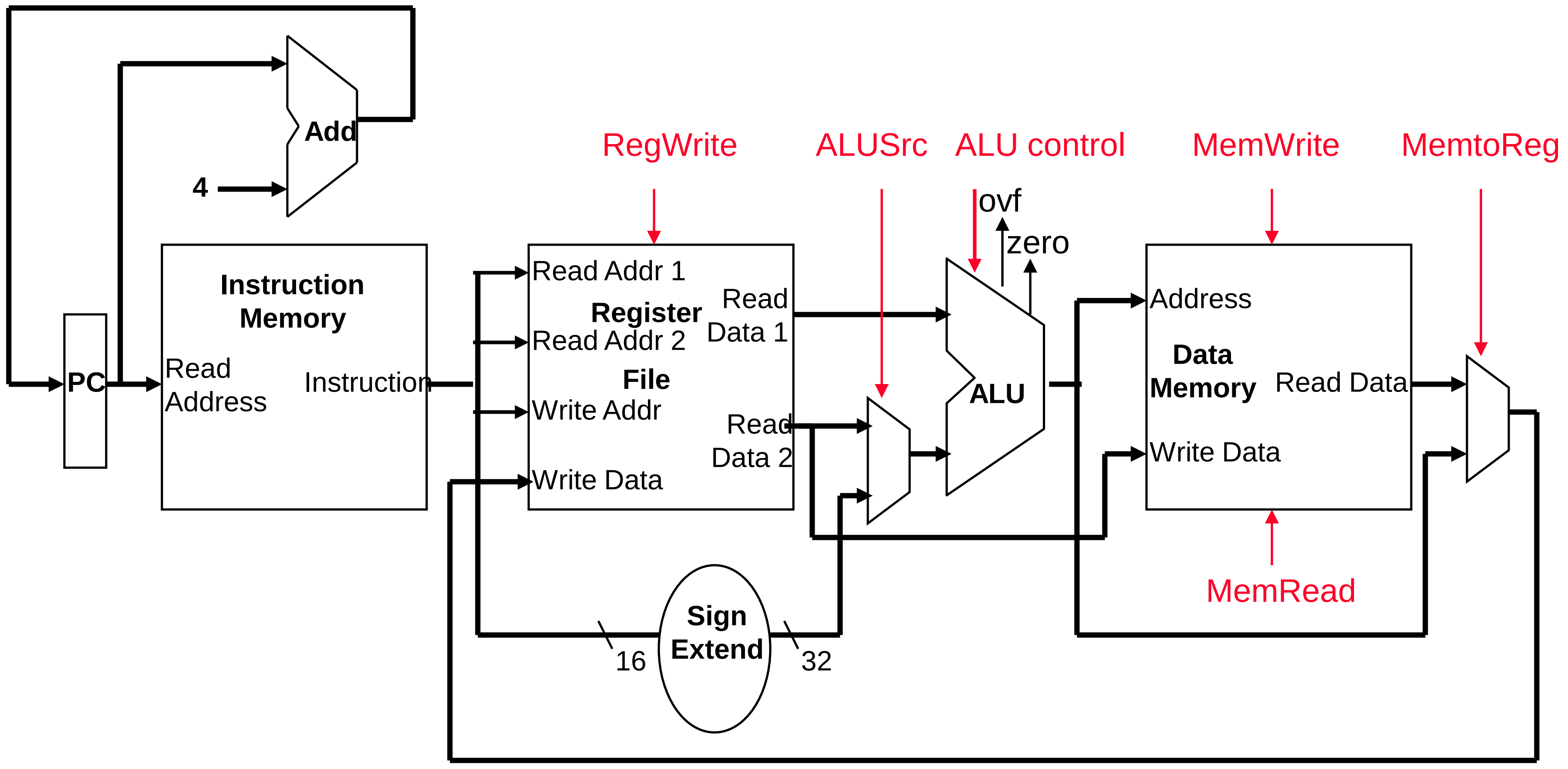
Creating a Single Datapath from the Parts

- Assemble the datapath segments and add control lines and multiplexors as needed

- Single cycle design – fetch, decode and execute each instructions in one clock cycle
  - no datapath resource can be used more than once per instruction, so some must be duplicated (e.g., separate Instruction Memory and Data Memory, several adders)
  - multiplexors needed at the input of shared elements with control lines to do the selection
  - write signals to control writing to the Register File and Data Memory

- Cycle time is determined by length of the longest path

Fetch, R, and Memory Access Portions


Add
4
PC
Instruction Memory
Read Address
Instruction
RegWrite
Read Addr 1
Register
Read Addr 2
File
Write Addr
Write Data
Read Data 1
Read Data 2
ALUSrc
ALU control
ovf
zero
ALU
MemWrite
Address
Data Memory
Read Data
Write Data
MemRead
MemtoReg
Sign Extend
16
32