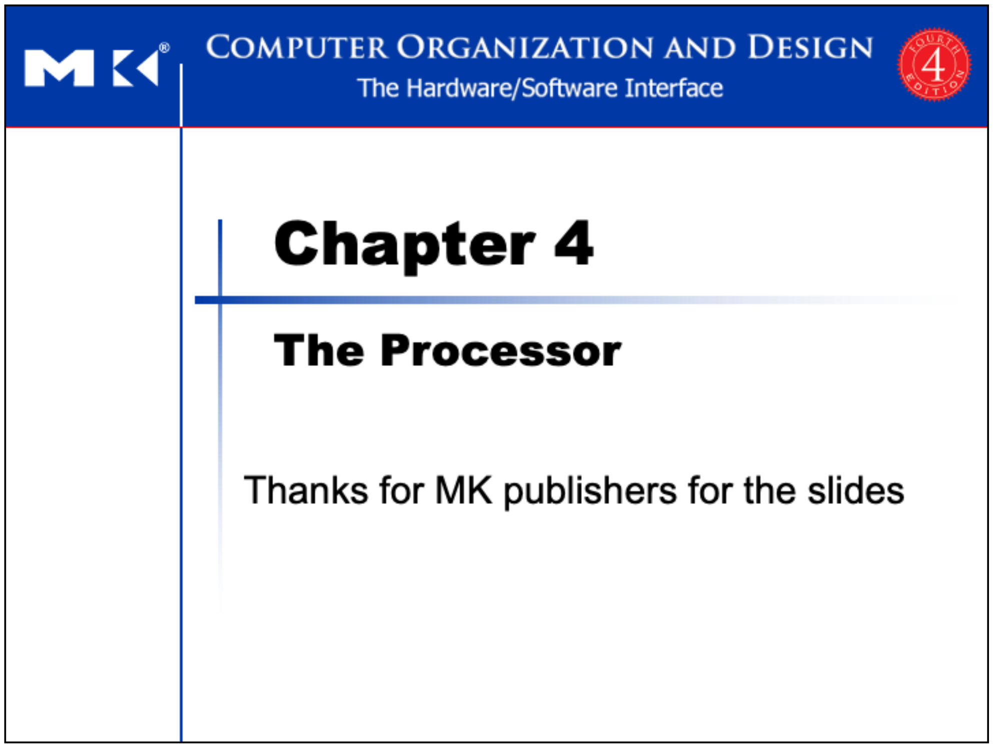
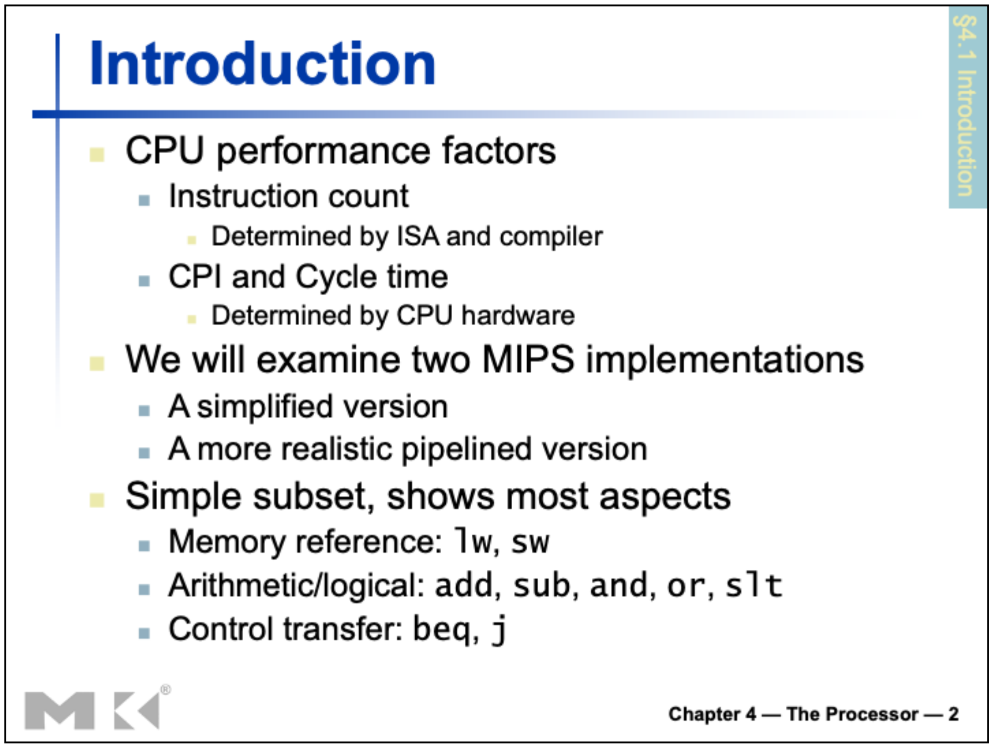


COMPUTER ORGANIZATION AND DESIGN
The Hardware/Software Interface

Chapter 4

The Processor

Thanks for MK publishers for the slides



Introduction

§4.1 Introduction

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine two MIPS implementations
  - A simplified version
  - A more realistic pipelined version
- Simple subset, shows most aspects
  - Memory reference: `lw`, `sw`
  - Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
  - Control transfer: `beq`, `j`

Chapter 4 — The Processor — 2

- Chapter 2 explains that the compiler and the instruction set architecture determine the instruction count of a given program. It includes the integer arithmetic-logical instructions, the memory-reference instructions, and the branch instructions.
- Chapter 3 explains how cycle per instruction is determined by CPU hardware – remember there are three possible implementations of multiplication, each takes different number of CPU clocks/cycles as the hardware is different

- This chapter contains an explanation of the **principles and techniques used in implementing a processor**, starting with a highly abstract and simplified overview

- We will be examining an implementation that includes a subset of the core MIPS instruction set:

  - The memory-reference instructions *load word* (lw) and *store word* (sw)

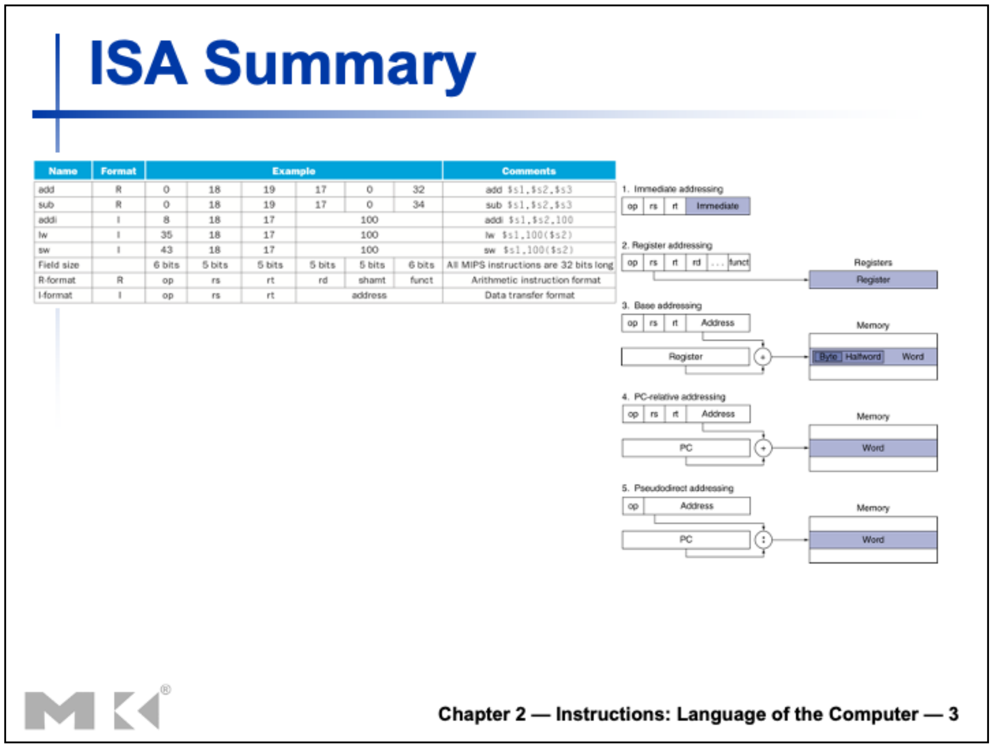
  - The arithmetic-logical instructions add, sub, AND, OR, and slt

  - The instructions *branch equal* (beq) and *jump* (j), which we add last

- This subset does not include all the integer instructions (for example, shift, multiply, and divide are missing), nor does it include any floating-point instructions.

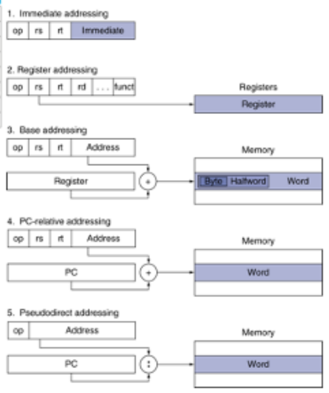
- In examining the implementation, we will have the opportunity to see how the instruction set architecture determines many aspects of the implementation, and
- how the choice of various implementation strategies affects the clock rate and CPI for the computer.

- Most concepts used to implement the MIPS subset in this chapter are the same basic ideas that are used to construct a broad spectrum of computers, from high-performance servers to general-purpose microprocessors to embedded processors.



ISA Summary

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
addi	I	8	18	17	100			addi $s1,$s2,100
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format



Chapter 2 — Instructions: Language of the Computer — 3

The MIPS addressing modes are the following:

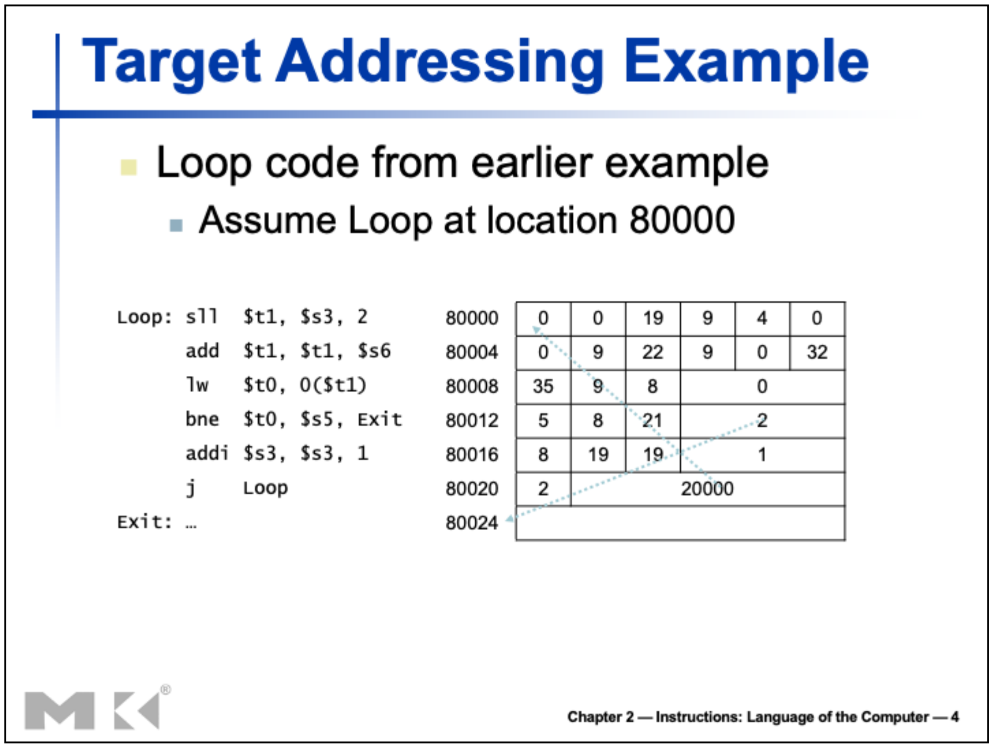
1. *Immediate addressing,* where the operand is a constant within the instruction itself (e.g., addi)

2. *Register addressing,* where the operand is a register (e.g., add, sub)
3. *Base* or *displacement addressing,* where the operand is at the memory location (e.g., lw, sw)

whose address is the sum of a register and a constant in the instruction

4. *PC-relative addressing,* where the branch address is the sum of the PC and a constant in the instruction (e.g., bne, be, blt)

5. *Pseudodirect addressing,* where the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC (e.g., jump)



Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

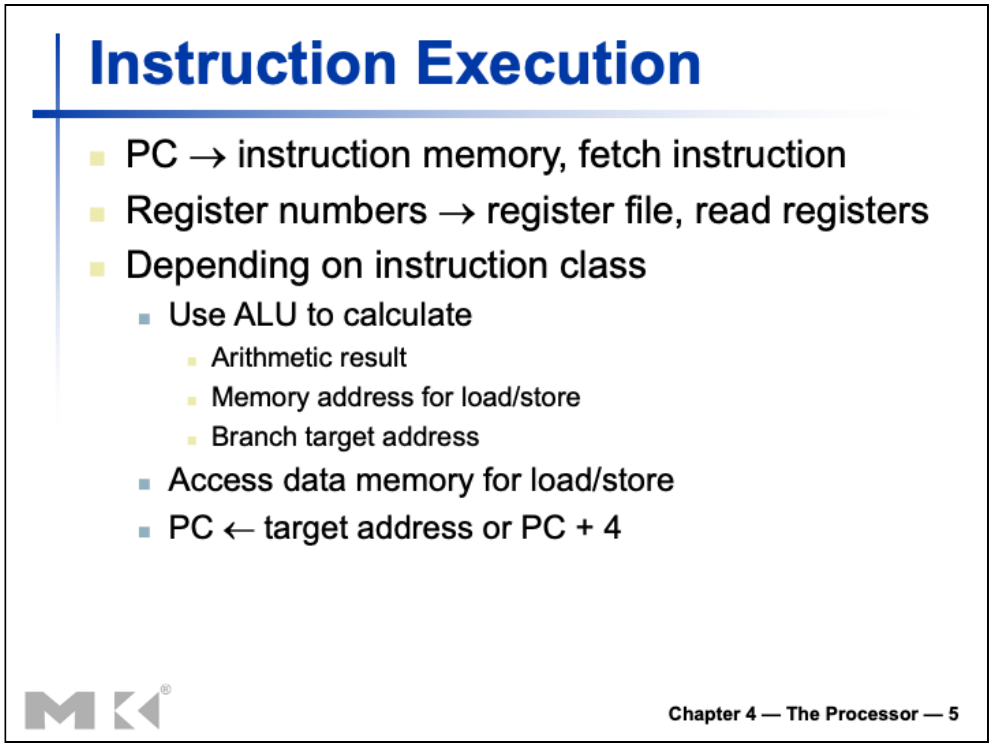
```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

80000	0	0	19	9	4	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024						

Chapter 2 — Instructions: Language of the Computer — 4

Remember that MIPS instructions have byte addresses, so addresses of sequential words differ by 4, the number of bytes in a word.

The bne instruction on the fourth line adds 2 words or 8 bytes to the address of the *following* instruction (80016), specifying the branch destination relative to that following instruction (8 + 80016) instead of relative to the branch instruction (12 + 80012) or using the full destination address (80024). The jump instruction on the last line does use the full address (20000 x 4 = 80000), corresponding to the label Loop.



Instruction Execution

- PC → instruction memory, fetch instruction
- Register numbers → register file, read registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch target address
  - Access data memory for load/store
  - PC ← target address or PC + 4

Chapter 4 — The Processor — 5

- Three classes of MIPS instructions, integer arithmetic-logical instructions, the memory-reference instructions, and the branch instructions.

- Much of what needs to be done to implement these instructions is the same, independent of the exact class of instruction. For every instruction, the first two steps are identical:

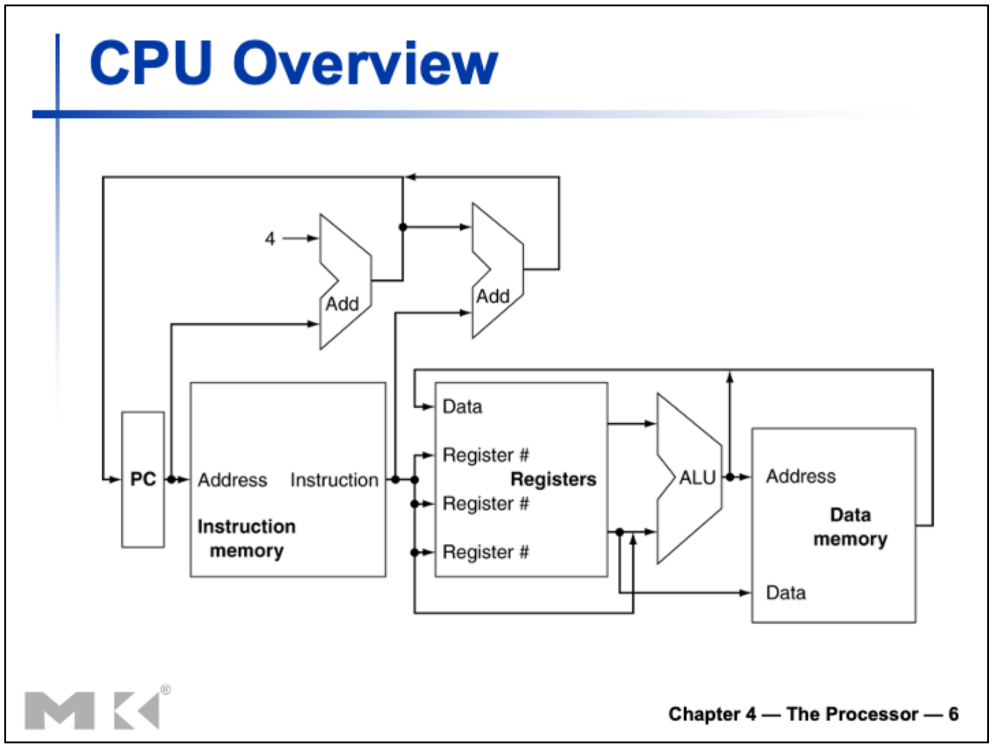
1. Send the *program counter* (PC) to the memory that contains the code and fetch the instruction from that memory.

2. Read one or two registers, using fields of the instruction to select the registers to read. For the load word instruction, we need to read only one register, but most other instructions require reading two registers.

- After these two steps, the actions required to complete the instruction depend on the instruction class. Fortunately, for each of the three instruction classes (memory-reference, arithmetic-logical, and branches), the actions are largely the same, independent of the exact instruction.

- all instruction classes, except jump, use the arithmetic-logical unit (ALU) after reading the registers.
  - the memory-reference instructions use the ALU for an address calculation,
  - the arithmetic-logical instructions for the operation execution, and

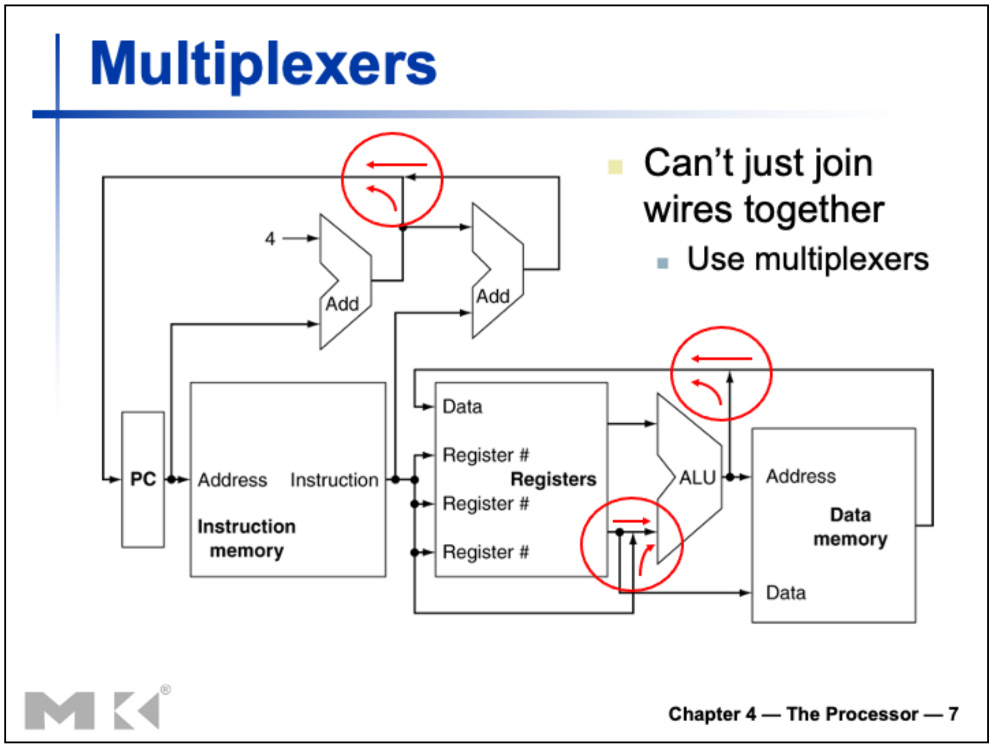
    - branches for comparison.
- After using the ALU, the actions required to complete various instruction classes differ.
    - A memory-reference instruction will need to access the memory either to read data for a load or write data for a store.
    - An arithmetic-logical must write the data from the ALU or load instruction write memory back into a register.
    - Lastly, for a branch instruction, we may need to change the next instruction address based on the comparison; otherwise, the PC should be incremented by 4 to get the address of the next instruction.



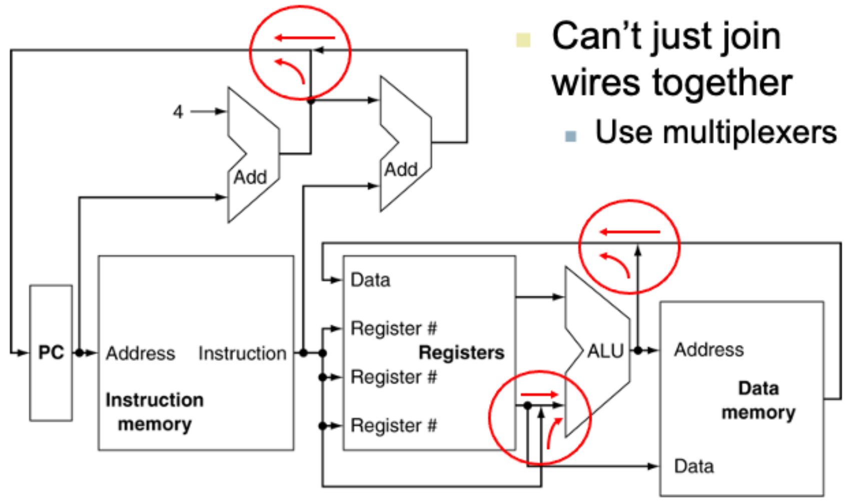
CPU Overview

- An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them.
- All instructions start by using the program counter to supply the instruction address to the instruction memory.
- After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction.
- Once the register operands have been fetched,
    - they can be operated on to compute a memory address (for a load or store),
    - to compute an arithmetic result (for an integer arithmetic-logical instruction),
    - or a compare (for a branch).
- If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register.
- If the operation is a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers.
- Branches require the use of the ALU output to determine the next instruction address, which comes either from comparsion result obtained from ALU (where the PC and branch offset are summed if branch result is true) or from an adder that increments the current PC by 4.
- The thick lines interconnecting the functional units represent buses, which

consist of multiple signals. The arrows are used to guide the reader in knowing how information flows. Since signal lines may cross, we explicitly show when crossing lines are connected by the presence of a dot where the lines cross.



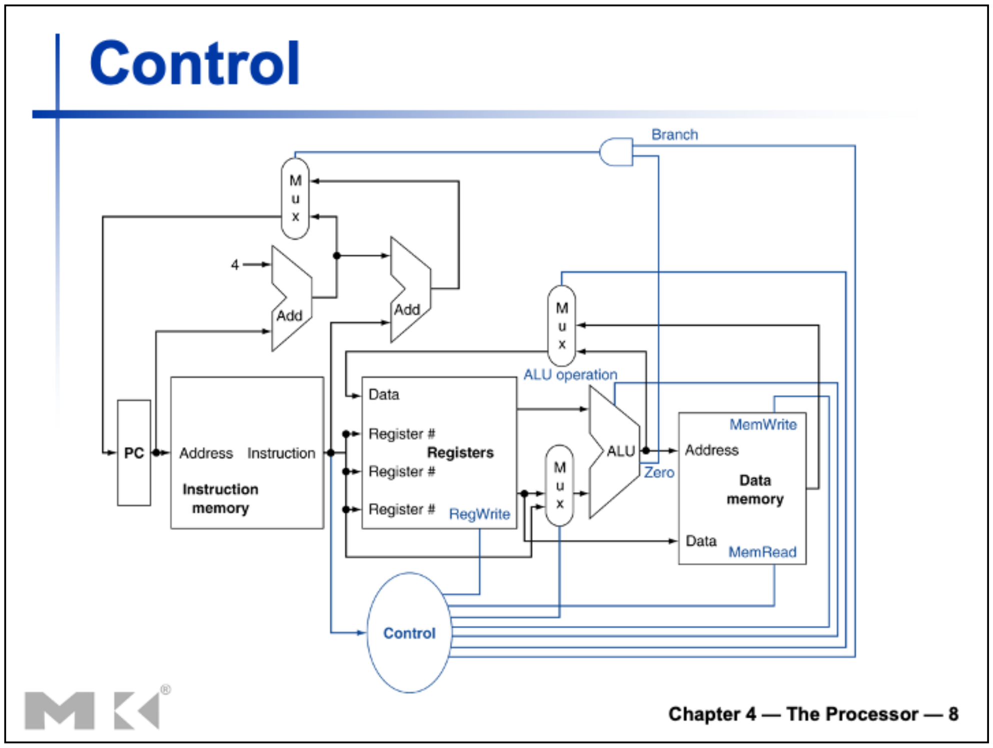
Multiplexers



- Can't just join wires together
  - Use multiplexers

Chapter 4 — The Processor — 7

- First, in several places, data going to a particular unit as coming from two different sources.
- For example,
  - the value written into the PC can come from one of two adders, (e.g., PC+4 or beq PC= X)
  - the second input to the ALU can come from a register or the immediate field of the instruction. (add vs addi,lw - one base reg and one offset address)
  - the data written into the register file can come from either the ALU or the data memory, (result can be sum of two regs for add or address at which data is present and should be loaded to a reg)
- In practice, these data lines cannot simply be wired together; we must add a logic element that chooses from among the multiple sources and steers one of those sources to its destination. This selection is commonly done with a device called a *multiplexor*, although this device might better be called a *data selector*.
- Multiplexer selects from among several inputs based on the setting of its control lines. The control lines are set based primarily on information taken from the instruction being executed.



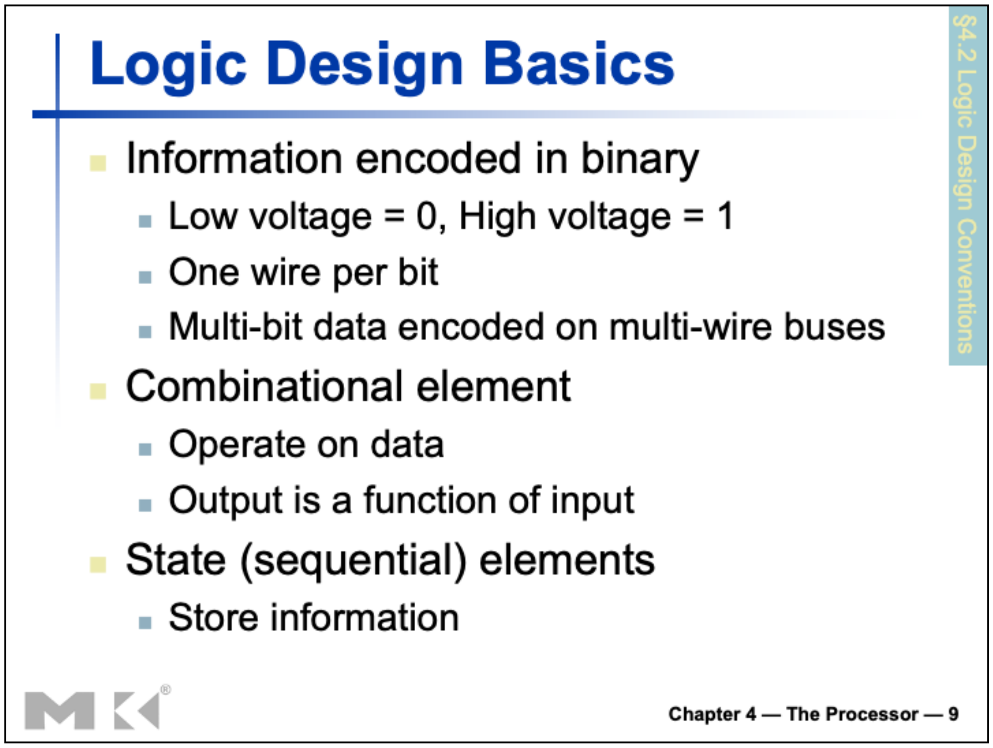
Control

Chapter 4 — The Processor — 8

- The basic implementation of the MIPS subset, including the necessary multiplexors and control lines.

- The top multiplexor ("Mux") controls what value replaces the PC (PC + 4 or the branch destination address in beq); the multiplexor is controlled by the gate that "ANDs" together the Zero output of the ALU and a control signal that indicates that the instruction is a branch.
- The middle multiplexor, whose output returns to the register file, is used to steer the output of the ALU (in the case of an arithmetic-logical instruction) or the output of the data memory (in the case of a load) for writing into the register file.
- Finally, the bottommost multiplexor is used to determine whether the second ALU input is from the registers (for an arithmetic-logical instruction or a branch in beq instruction) or from the offset field of the instruction (for a load or store).
- The added control lines are straightforward and determine the operation performed at the ALU,
  - whether the data memory should read or write, and
  - whether the registers should perform a write operation.
  - The control lines are shown in color to make them easier to see.

-Next we will describe a simple implementation that uses a single long clock cycle for every instruction and follows the general form of this figure. In this first design, every instruction begins execution on one clock edge and completes execution on the next clock edge.



Logic Design Basics

§4.2 Logic Design Conventions

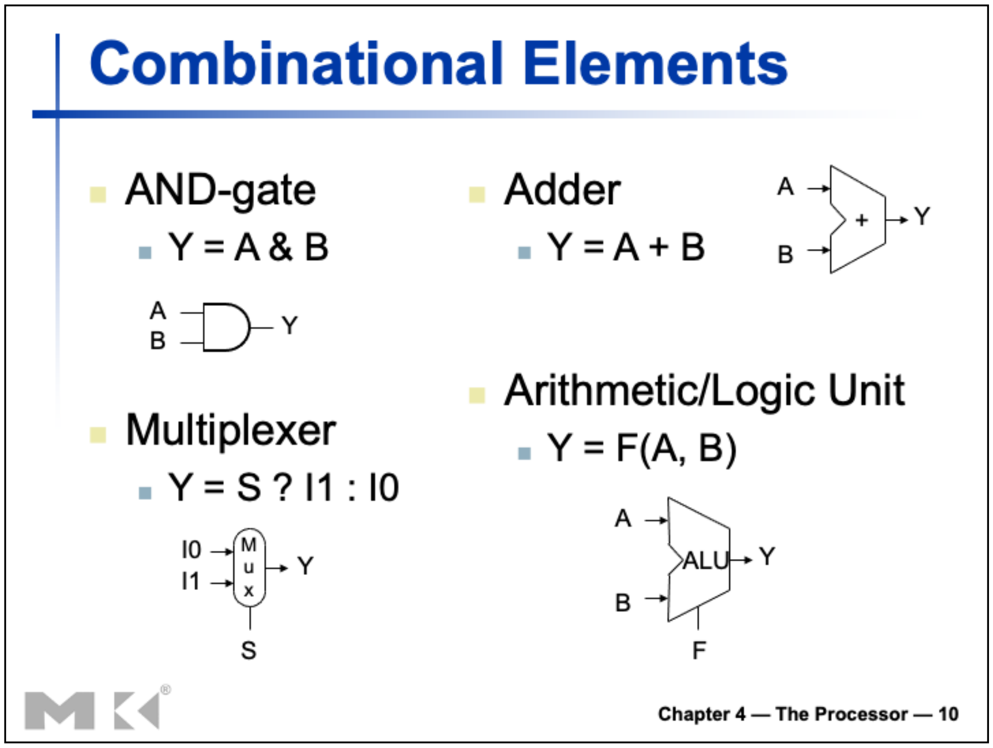
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Chapter 4 — The Processor — 9

- To discuss the design of a computer, we must decide how the hardware logic implementing the computer will operate and how the computer is clocked.
- Let's reviews a few key ideas in digital logic that we will use extensively in this chapter.

- The datapath elements in the MIPS implementation consist of two different types of logic elements:
  - elements that operate on data values and
  - elements that contain state, also called sequential elements

- The elements that operate on data values are all **combinational**, which means that their outputs depend only on the current inputs. Given the same input, a combinational element always produces the same output. Given a set of inputs, it always produces the same output because it has no internal storage.
- Other elements in the design are not combinational, but instead contain *state*.

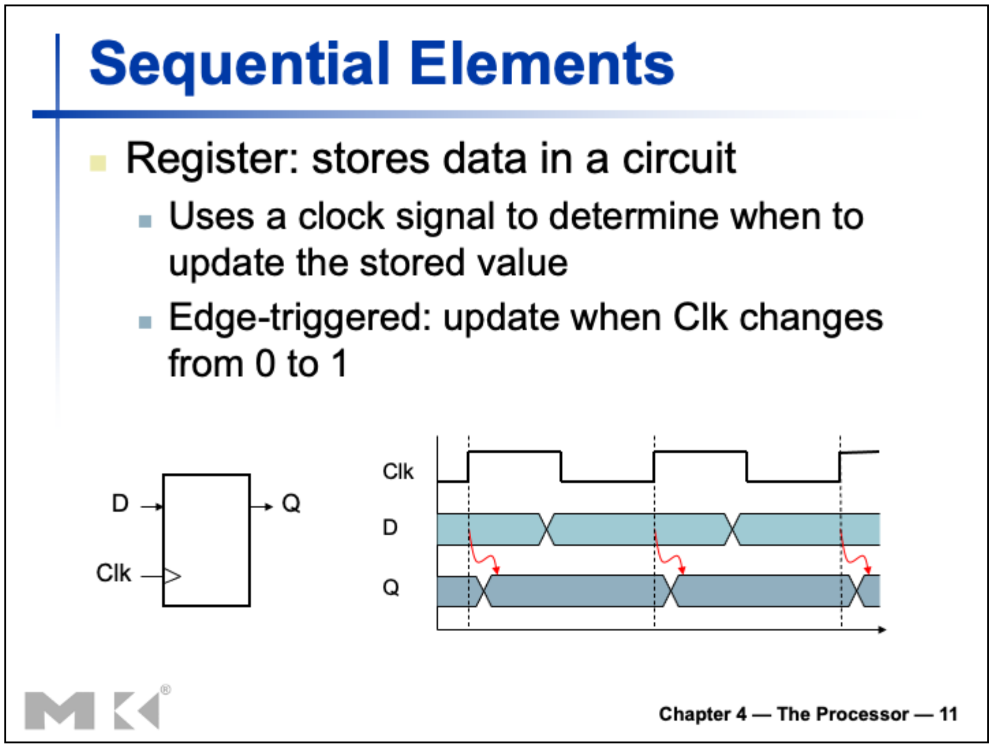


Combinational Elements

- AND-gate
  - Y = A & B
- Multiplexer
  - Y = S ? I1 : I0
- Adder
  - Y = A + B
- Arithmetic/Logic Unit
  - Y = F(A, B)

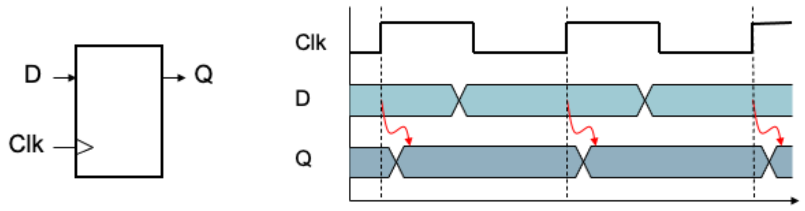
A
B
Y
I0
I1
M
u
x
S
+
ALU
F
Chapter 4 — The Processor — 10

- **combinational element:** An operational element, such as an AND gate or an ALU. their outputs depend only on the current inputs. Given the same input, a combinational element always produces the same output.
- Given a set of inputs, it always produces the same output because it has no internal storage.



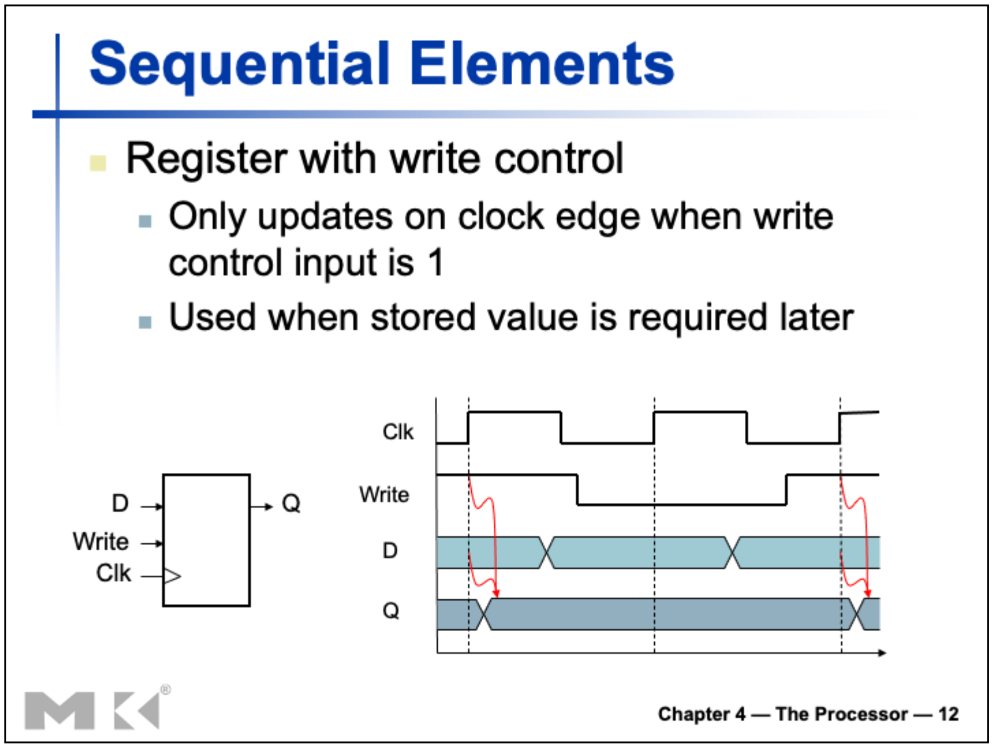
Sequential Elements

- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1



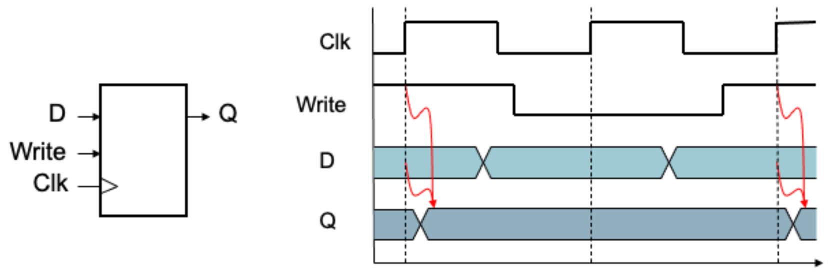
Chapter 4 — The Processor — 11

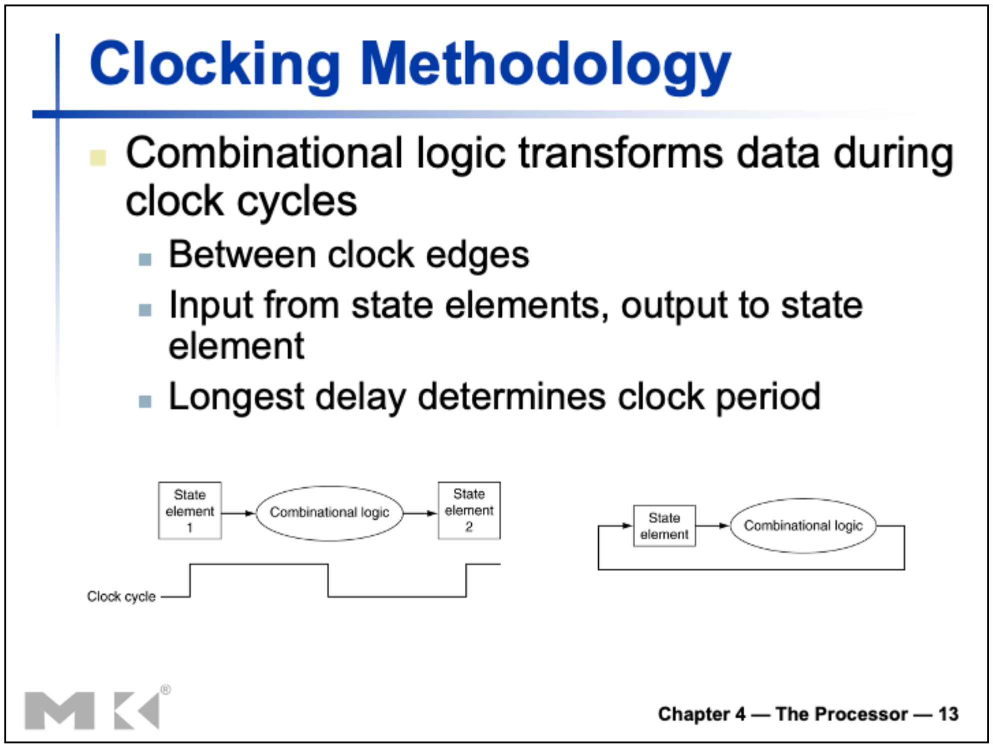
- state elements are also called *sequential elements*
- An element contains state if it has some internal storage.
- We call these elements **state elements** because, if we pulled the power plug on the computer, we could restart it accurately by loading the state elements with the values they contained before we pulled the plug. Furthermore, if we saved and restored the state elements, it would be as if the computer had never lost power. Thus, these state elements completely characterize the computer.
- D-flipflop, instruction and data memories, and registers, are all examples of state elements.
- A state element has at least two inputs and one output.
  - The required inputs are the data value to be written into the element
  - The clock is used to determine when the state element should be written; a state element can be read at any time.
  - The output from a state element provides the value that was written in an earlier clock cycle.
  - Ex: Register is a state element, that can be both read and written on the same clock cycle,



Sequential Elements

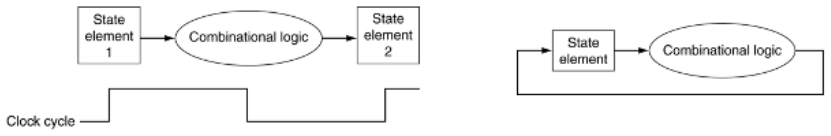
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later





Clocking Methodology

- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period



Chapter 4 — The Processor — 13

- **clocking methodology:** The approach used to determine when data is valid and stable relative to the clock.
  - A **clocking methodology** defines when signals can be read and when they can be written. It is important to specify the timing of reads and writes, because if a signal is written at the same time it is read, the value of the read could correspond to the old value, the newly written value, or even some mix of the two!

- **edge-triggered clocking:** A clocking scheme in which all state changes occur on a clock edge.
  - In an edge-triggered design, either the rising or falling edge of the clock is active and causes state to be changed.
  - the *clock cycle time* or *clock period* is divided into two portions: when the clock is high and when the clock is low.
  - In the lectures, we use only **edge-triggered clocking**. This means that all state changes occur on a clock edge.

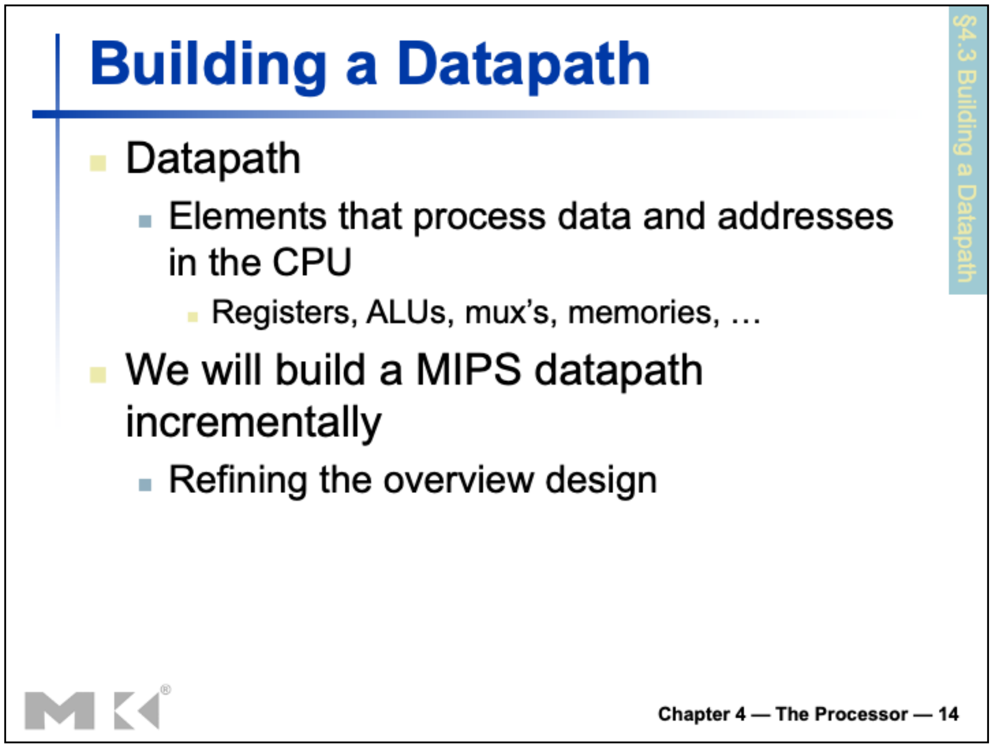
- combinational logic must have its inputs come from a set of state elements and its outputs written into a set of state elements.
- Left figure: shows the two state elements surrounding a block of combinational logic, which operates in a single clock cycle: all signals must

propagate from state element 1, through the combinational logic, and to state element 2 in the time of one clock cycle. The time necessary for the signals to reach state element 2 defines the length of the clock cycle.

  - Ex: Inputs: register 1, register 2. Combination logic: and gate, Output: Register 3

- Right figure: An edge-triggered methodology allows us to read the contents of a register, send the value through some combinational logic, and write that register in the same clock cycle.

- For the 32-bit MIPS architecture, nearly all of these state and logic elements will have inputs and outputs that are 32 bits wide, since that is the width of most of the data handled by the processor.



Building a Datapath

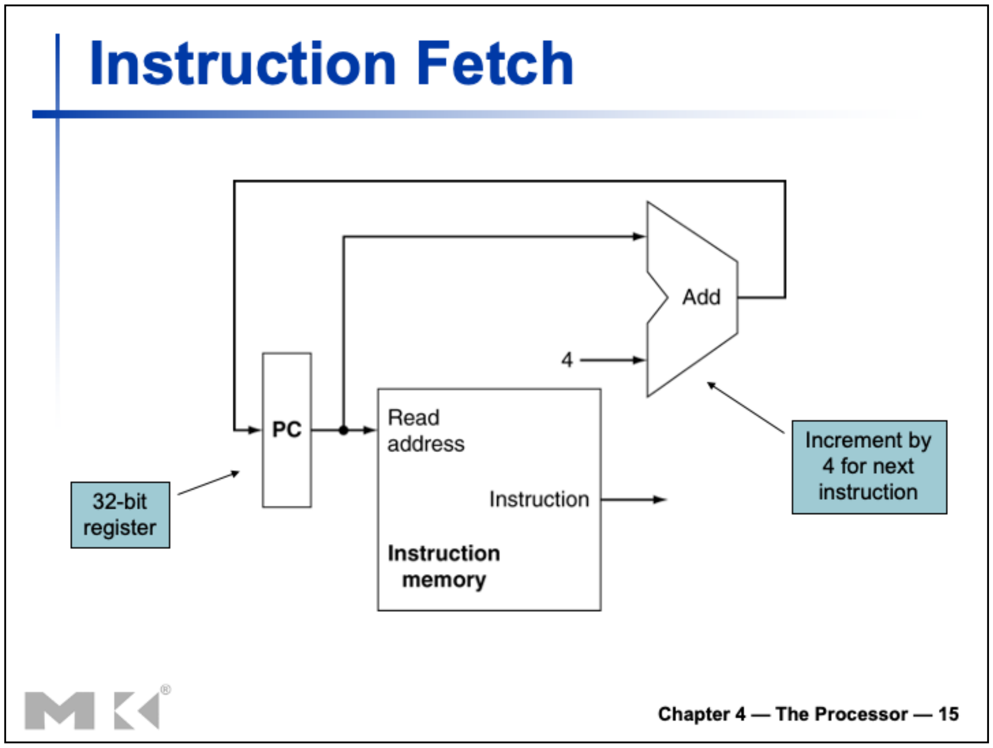
- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- We will build a MIPS datapath incrementally
  - Refining the overview design

§4.3 Building a Datapath

Chapter 4 — The Processor — 14

- Let's start a datapath design is to examine the major components required to execute each class of MIPS instructions.

- Let's start at the top by looking at which **datapath elements** each instruction needs, and then work our way down through the levels of **abstraction**.



Instruction Fetch

Chapter 4 — The Processor — 15

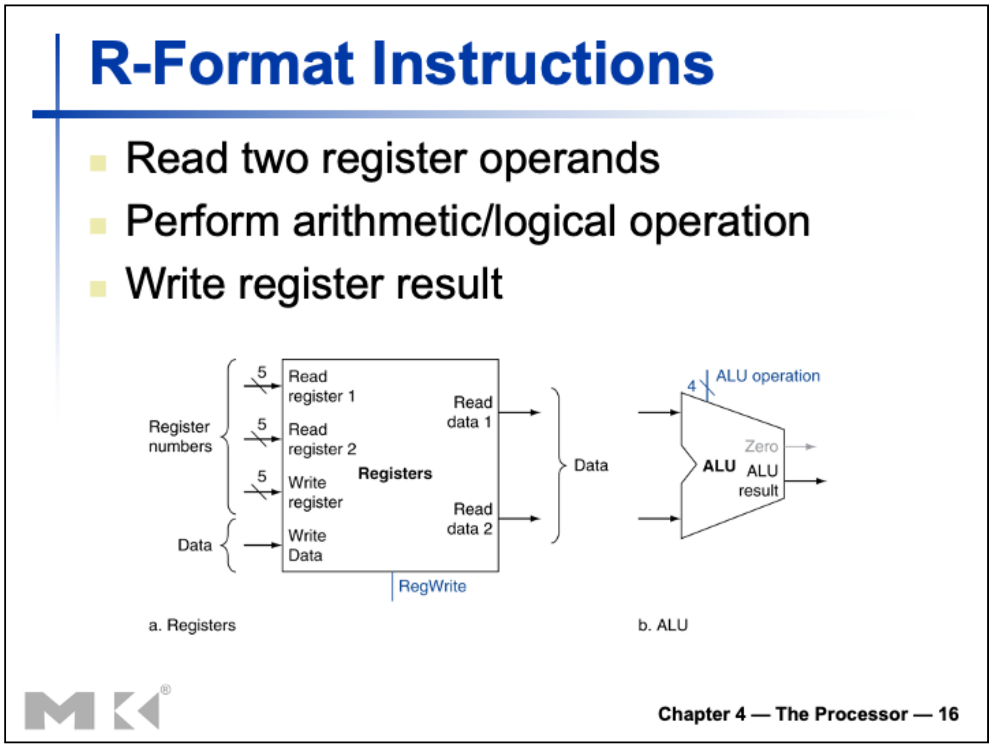
- let's look at Datapath elements require to execute an instruction and prepare which instruction to execute next.

- To execute any instruction, we must start by fetching the instruction from memory. To prepare for executing the next instruction, we must also increment the program counter so that it points at the next instruction, 4 bytes later. The figure shows how to combine the three elements to form a datapath that fetches instructions and increments the PC to obtain the address of the next sequential instruction.

- Two state elements are the instruction memory and the program counter (register). Both are needed to store and access instructions. In addtiion, an adder (comb element) is needed to compute the next instruction address.
    - The instruction memory need only provide read access because the datapath does not write instructions.
    - Since the instruction memory only reads, we treat it as combinational logic: the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed. (We will need to write the instruction memory when we load the program; this is not hard to add, and we ignore it for simplicity.)
    - The program counter is a 32-bit register that is written at the end of every clock cycle and thus does not need a write control signal.
    - The adder is an ALU wired to always add its two 32-bit inputs and

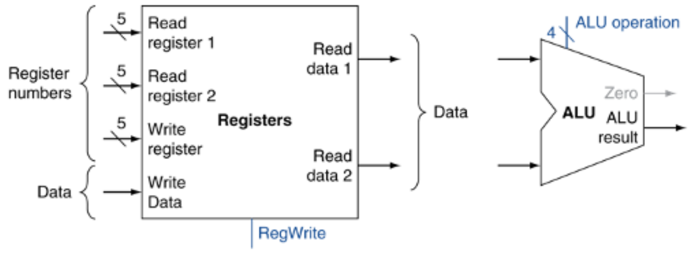
place the sum on its output.

- Read PC → apply combination logic (access memory location & adder) → update PC
- How many clock cycles do you need? 1!!



R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result



a. Registers b. ALU

Chapter 4 — The Processor — 16

- The two elements needed to implement R-format ALU operations are the register file and the ALU.

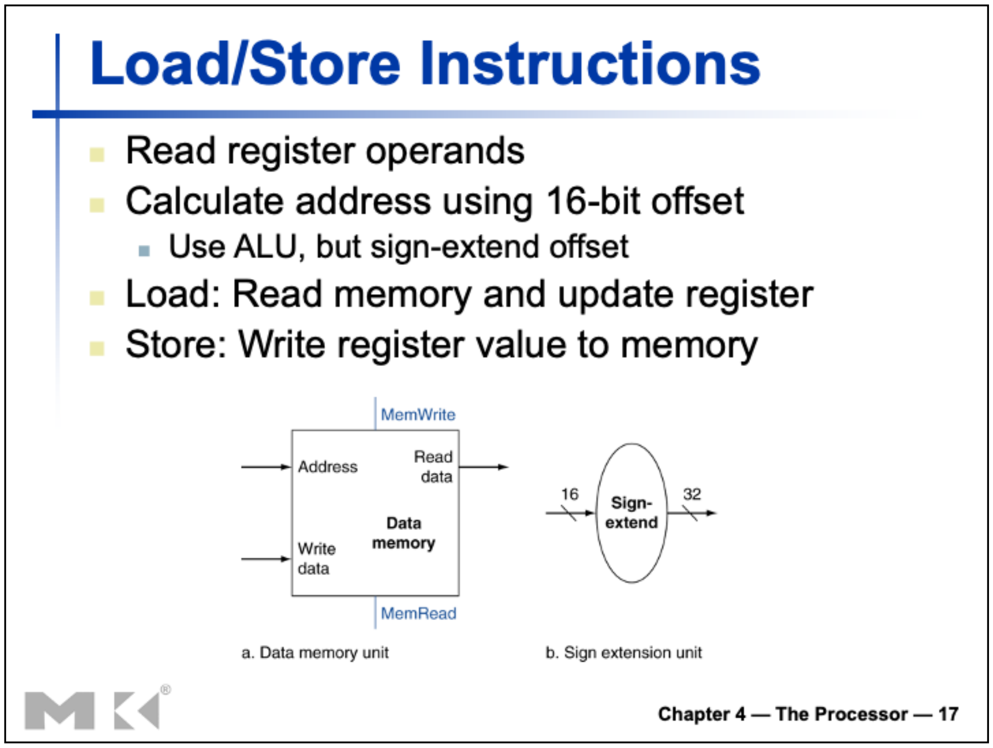
- The processor's 32 general-purpose registers are stored in a structure called a **register file**. A register file is a collection of registers in which any register can be read or written by specifying the number of the register in the file.

- R-format instructions have three register operands, so we will need to read two data words from the register file and write one data word into the register file for each instruction.

- For each data word to be read from the registers, we need an input to the register file that specifies the *register number* to be read and an output from the register file that will carry the value that has been read from the registers.
- To write a data word, we will need two inputs: one to specify the register number to be written and one to supply the *data* to be written into the register.
- The register file always outputs the contents of whatever register numbers are on the Read register inputs.
- Writes, however, are controlled by the write control signal, which must be asserted for a write to occur at the clock edge.

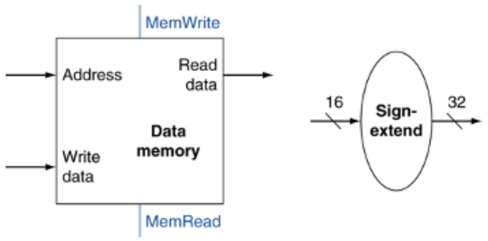
- Figure left, shows the result; we need a total of four inputs (three for register numbers and one for data) and two outputs (both for data).
- The register number inputs are 5 bits wide to specify one of 32 registers (32 = 2^5), whereas the data input and two data output buses are each 32 bits wide.
- Figure right, shows the ALU, which takes two 32-bit inputs and produces a 32-bit result, as well as a 1-bit signal if the result is 0. The 4-bit control signal of the ALU is described in detail later.

- To summarize, the register file contains all the registers and has two read ports and one write port. The design of multiported register files is discussed in Section B.8 of **Appendix B**. The register file always outputs the contents of the registers corresponding to the Read register inputs on the outputs; no other control inputs are needed. In contrast, a register write must be explicitly indicated by asserting the write control signal. Remember that writes are edge-triggered, so that all the write inputs (i.e., the value to be written, the register number, and the write control signal) must be valid at the clock edge. Since writes to the register file are edge-triggered, our design can legally read and write the same register within a clock cycle: the read will get the value written in an earlier clock cycle, while the value written will be available to a read in a subsequent clock cycle.



Load/Store Instructions

- Read register operands
- Calculate address using 16-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory

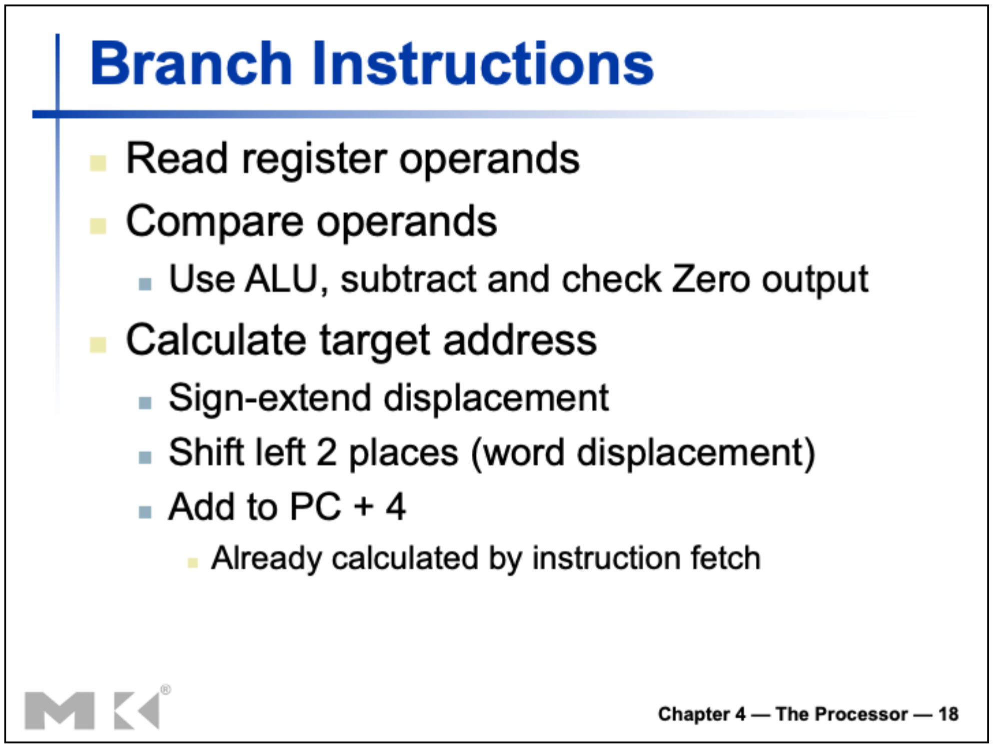


a. Data memory unit b. Sign extension unit

Chapter 4 — The Processor — 17

- Next, consider the MIPS load word and store word instructions, which have the general form lw $t1,offset_value($t2) or sw $t1,offset_value ($t2).
  - These instructions compute a memory address by adding the base register, which is $t2, to the 16-bit signed offset field contained in the instruction.
  - If the instruction is a store, the value to be stored must also be read from the register file where it resides in $t1.
  - If the instruction is a load, the value read from memory must be written into the register file in the specified register, which is $t1.
  - Thus, we will need both the register file and the ALU in the previous slide.
  - In addition,
    - we will need a unit to **sign-extend** the 16-bit offset field in the instruction to a 32-bit signed value, and
    - a data memory unit to read from or write to. The data memory must be written on store instructions; hence, data memory has read and write control signals, an address input, and an input for the data to be written into memory. The figure left, shows these two elements.
- To summarize, the two units needed to implement loads and stores, in addition

to the register file and ALU in previous slide, are the data memory unit and the sign extension unit.

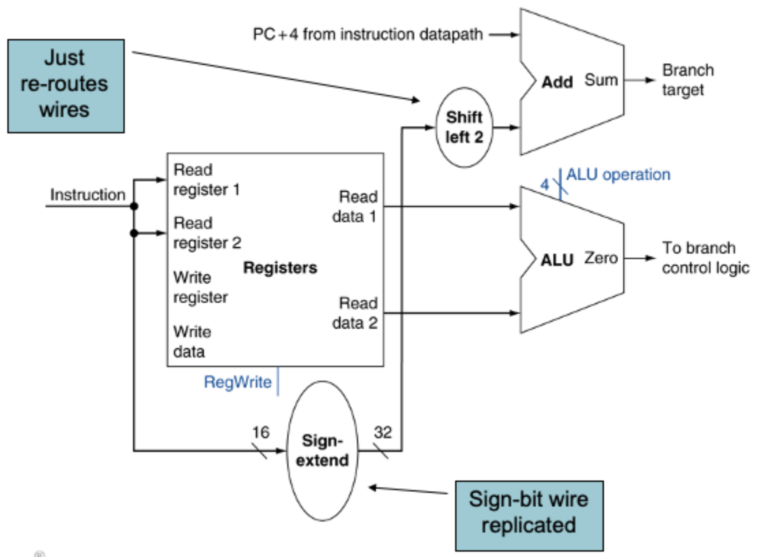


Branch Instructions

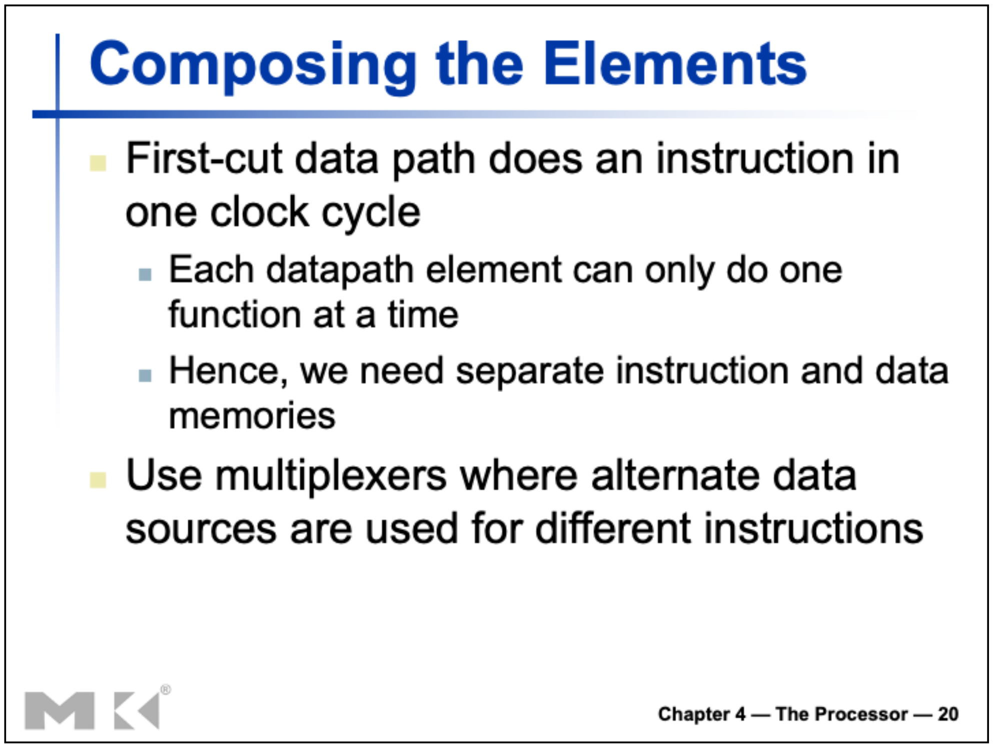
- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated by instruction fetch

Chapter 4 — The Processor — 18

Branch Instructions



Chapter 4 — The Processor — 19



Composing the Elements

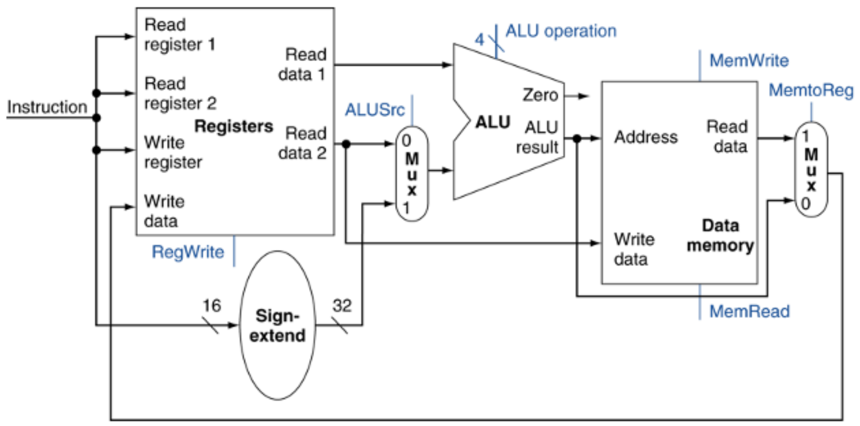
- First-cut data path does an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

Chapter 4 — The Processor — 20

- This simplest datapath will attempt to execute all instructions in one clock cycle.
- This means that no datapath resource can be used more than once per instruction, so any element needed more than once must be duplicated.
- We therefore need a memory for instructions separate from one for data.

- To share a datapath element between two different instruction classes, we may need to allow multiple connections to the input of an element, using a multiplexor and control signal to select among the multiple inputs.

R-Type/Load/Store Datapath

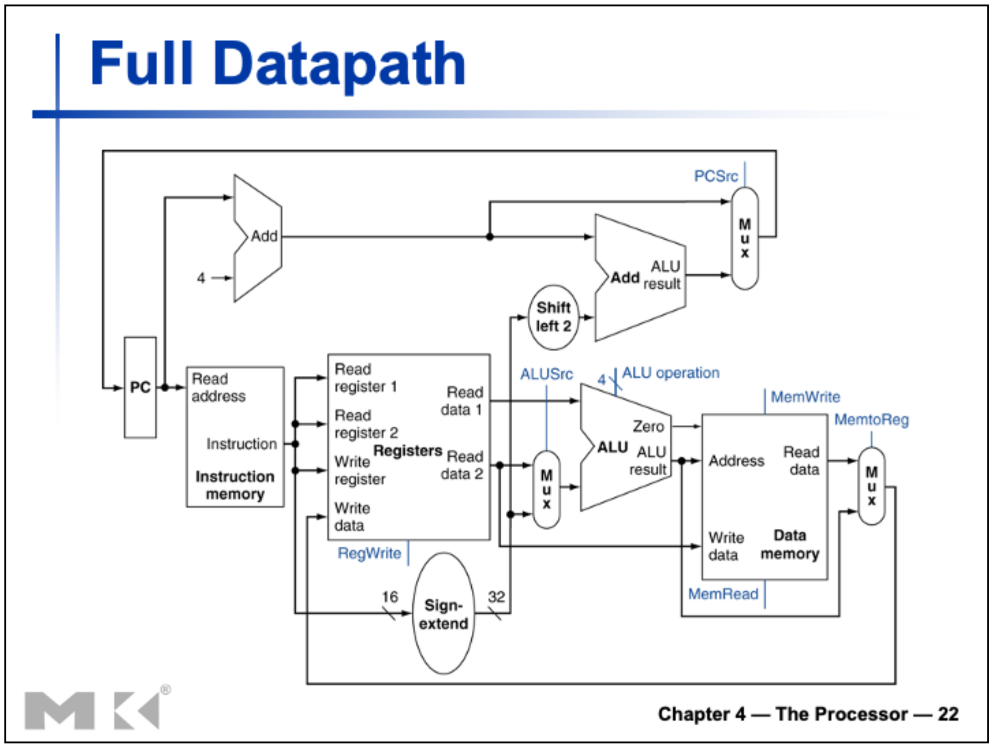


Chapter 4 — The Processor — 21

- The operations of arithmetic-logical (or R-type) instructions and the memory instructions datapath are quite similar.
- The key differences are the following:
    - The arithmetic-logical instructions use the ALU, with the inputs coming from the two registers. The memory instructions can also use the ALU to do the address calculation, although the second input is the sign- extended 16-bit offset field from the instruction.
    - The value stored into a destination register comes from the ALU (for an R-type instruction) or the memory (for a load).

Show how to build a datapath for the operational portion of the memory-reference and arithmetic-logical instructions that uses a single register file and a single ALU to handle both types of instructions, adding any necessary multiplexors.

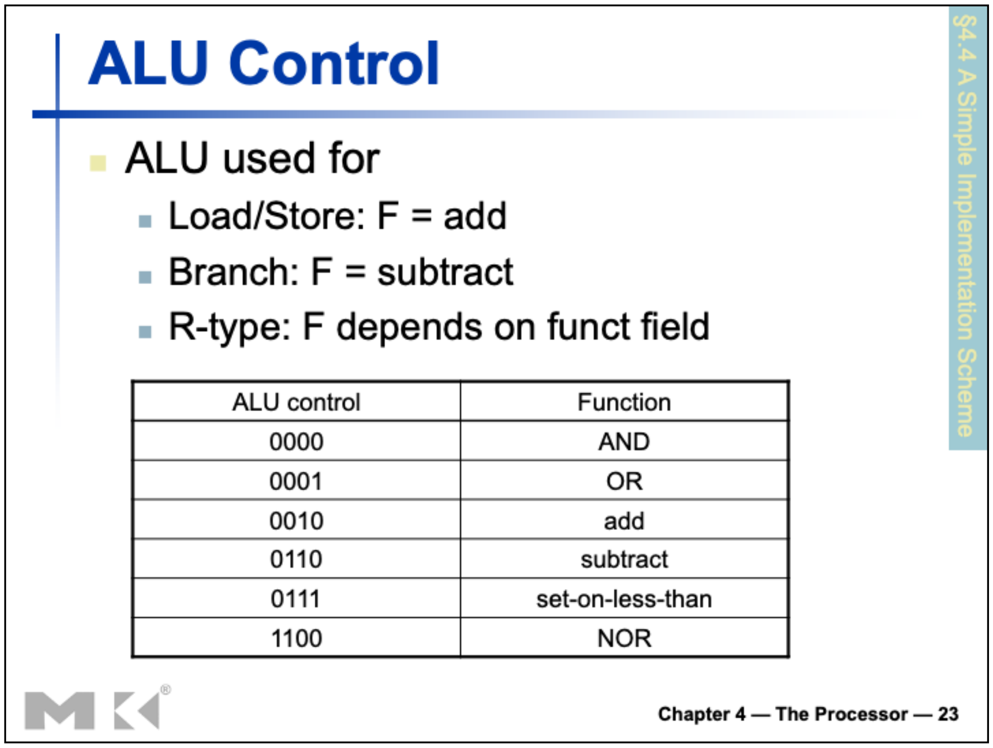
To create a datapath with only a single register file and a single ALU, we must support two different sources for the second ALU input, as well as two different sources for the data stored into the register file. Thus, one multiplexor is placed at the ALU input and another at the data input to the register file.



Full Datapath

Chapter 4 — The Processor — 22

- Now we can combine all the pieces to make a simple datapath for the core MIPS architecture by adding the datapath for instruction fetch ,the datapath from R-type and memory instructions, and the datapath for branches.
- This figure shows the datapath we obtain by composing the separate pieces.
- The branch instruction uses the main ALU for comparison of the register operands, so we must keep the adder for computing the branch target address. An additional multiplexor is required to select either the sequentially following instruction address (PC + 4) or the branch target address to be written into the PC.
-



ALU Control

§4.4 A Simple Implementation Scheme

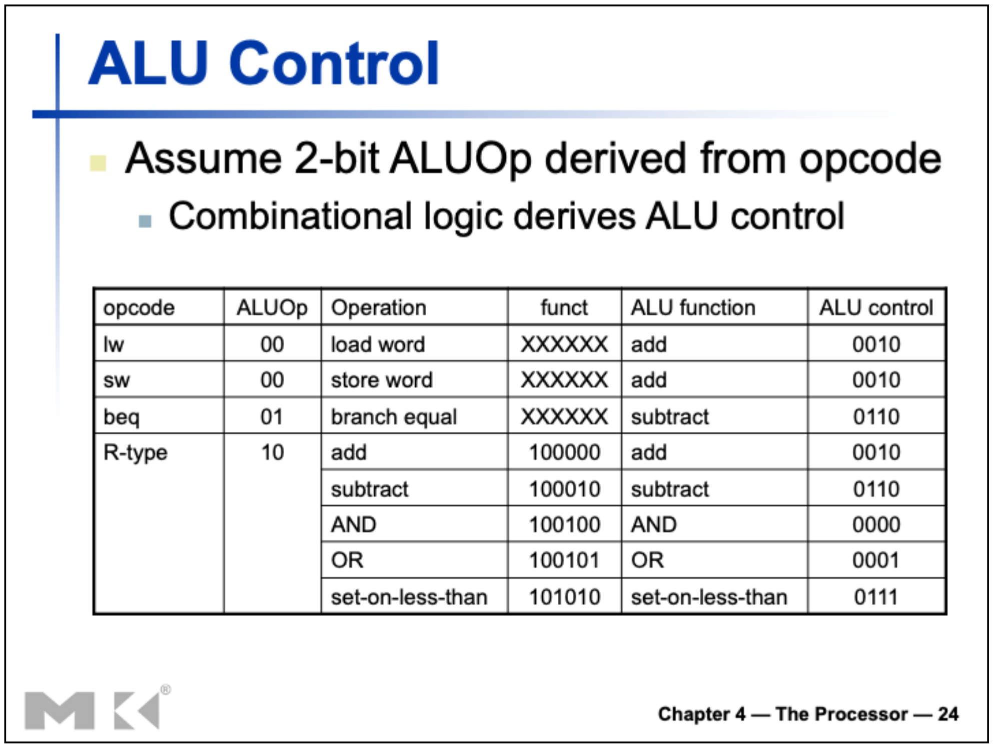
- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on funct field

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

Chapter 4 — The Processor — 23

- Now that we have completed this simple datapath, we can add the control unit. The control unit must be able to take inputs and generate a write signal for each state element, the selector control for each multiplexor, and the ALU control.

- Let's discuss ALU control first:
- We can generate the 4-bit ALU control input using a small control unit that has as inputs the function field of the instruction and a 2-bit control field, which we call ALUOp.
  - ALUOp (2bit): ALUop indicates whether the operation to be performed should be add (00) for loads and stores, subtract (01) for beq, or determined by the operation encoded in the funct field (10)
  - Funct field (4bit): Function field of the instruction
  - ALU control inpute (4bit) : one of the five 4-bit combinations above



ALU Control

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

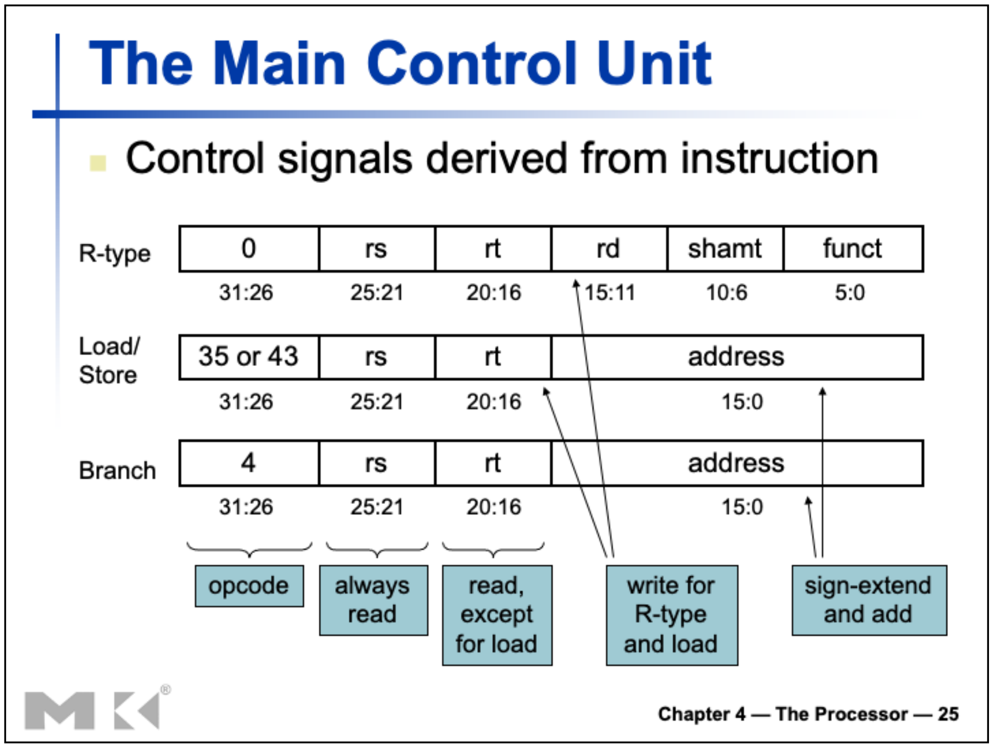
opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

Chapter 4 — The Processor — 24

- show how to set the ALU control inputs based on the 2-bit ALUOp control and the 6-bit function code.

- How the ALU control bits are set depends on the ALUOp control bits and the different function codes for the R-type instruction.

- The opcode, listed in the first column, determines the setting of the ALUOp bits. All the encodings are shown in binary.
- Notice that when the ALUOp code is 00 or 01, the desired ALU action does not depend on the function code field; in this case, we say that we "don't care" about the value of the function code, and the funct field is shown as XXXXXX.
- When the ALUOp value is 10, then the function code is used to set the ALU control input.



The Main Control Unit

- Control signals derived from instruction

- The three instruction classes (R-type, load and store, and branch) use two different instruction formats.

(a) Instruction format for R-format instructions, which all have an opcode of 0. These instructions have three register operands: rs, rt, and rd. Fields rs and rt are sources, and rd is the destination. The ALU function is in the funct field and is decoded by the ALU control design in the previous slide. The R-type instructions that we implement are add, sub, AND, OR, and slt. The shamt field is used only for shifts; we will ignore it in this chapter.

(b) Instruction format for load (opcode = 35ten) and store (opcode = 43ten) instructions. The register rs is the base register that is added to the 16-bit address field to form the memory address. For loads, rt is the destination register for the loaded value. For stores, rt is the source register whose value should be stored into memory.

(c) Instruction format for branch equal (opcode =4). The registers rs and rt are the source registers that are compared for equality. The 16-bit address field is sign-extended, shifted, and added to the PC + 4 to compute the branch target address

There are several major observations about this instruction format that we will rely on:

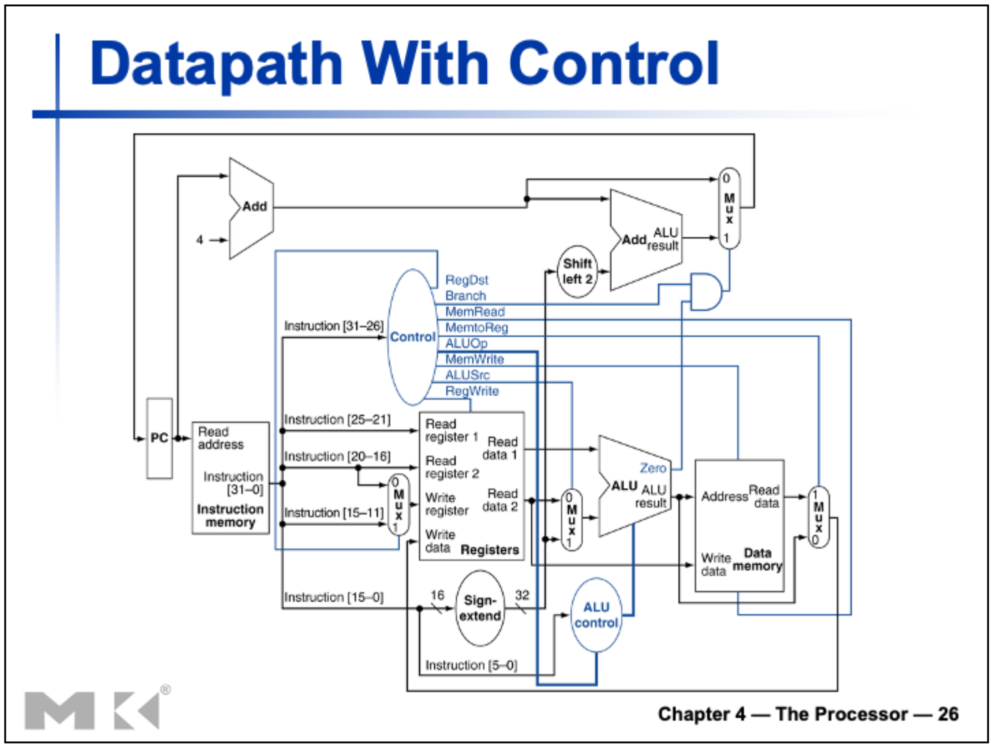
■ The op field, which as we saw in Chapter 2 is called the **opcode**, is always contained in bits 31:26. We will refer to this field as Op[5:0].

■ The two registers to be read are always specified by the rs and rt fields, at positions 25:21 and 20:16. This is true for the R-type instructions, branch equal, and store.

■ The base register for load and store instructions is always in bit positions 25:21 (rs).

■ The 16-bit offset for branch equal, load, and store is always in positions 15:0.

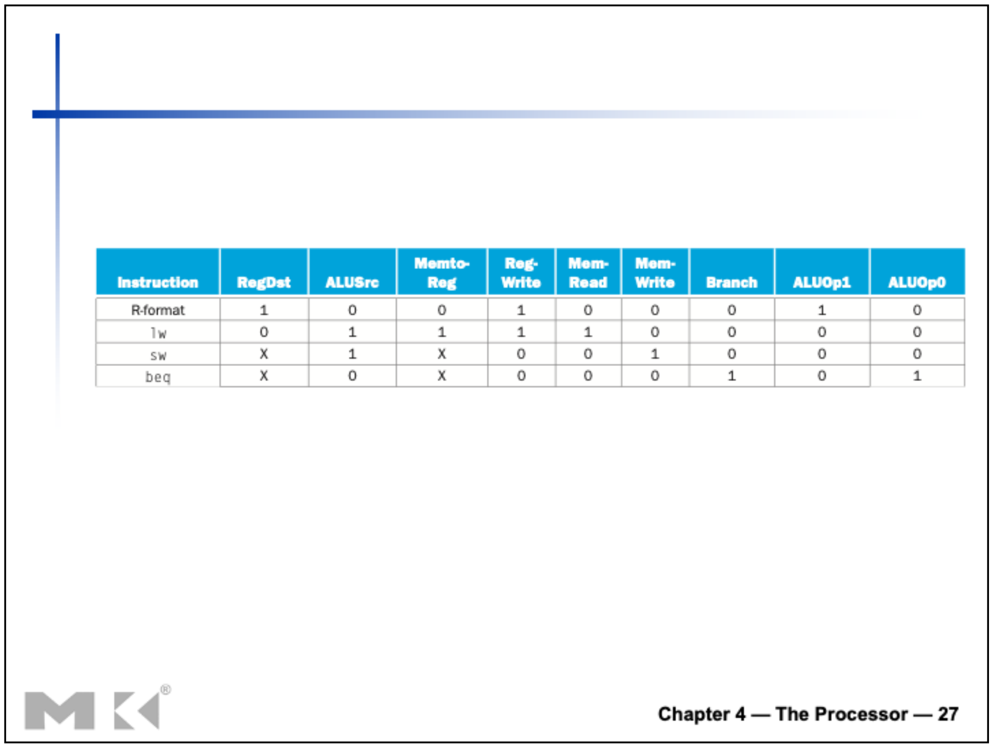
■ The destination register is in one of two places. For a load it is in bit positions 20:16 (rt), while for an R-type instruction it is in bit positions 15:11 (rd). Thus, we will need to add a multiplexor to select which field of the instruction is used to indicate the register number to be written.



Datapath With Control

Chapter 4 — The Processor — 26

- The simple datapath with the control unit.
- The input to the control unit is the 6-bit opcode field from the instruction.
- There are total nine control signals as output. The outputs of the control unit consist of
  - three 1-bit signals that are used to control multiplexors (RegDst, ALUSrc, and MemtoReg),
  - three signals for controlling reads and writes in the register file and data memory (RegWrite, MemRead, and MemWrite),
  - a 1-bit signal used in determining whether to possibly branch (Branch), and
  - a 2-bit control signal for the ALU (ALUOp).
- An AND gate is used to combine the branch control signal and the Zero output from the ALU; the AND gate output controls the selection of the next PC.



Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

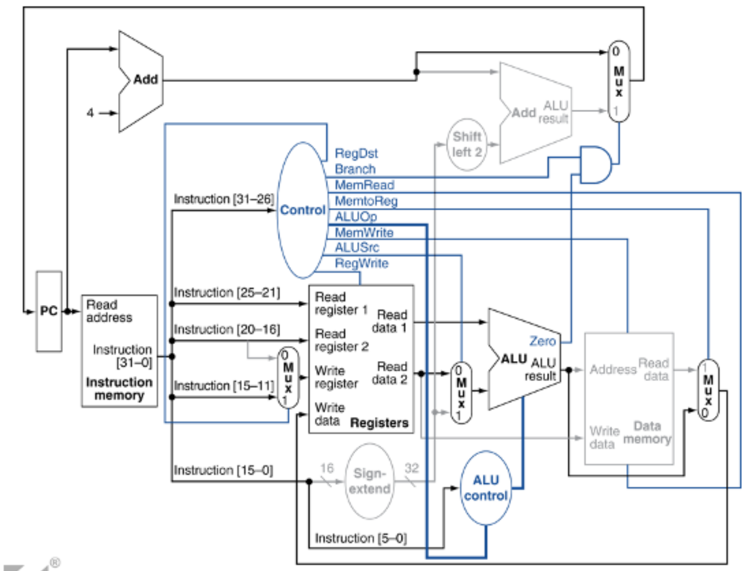
Chapter 4 — The Processor — 27

- The setting of the control lines is completely determined by the opcode fields of the instruction.

- The first row of the table corresponds to the R-format instructions (add, sub, AND, OR, and slt).
    - For all these instructions, the source register fields are rs and rt, and the destination register field is rd; this defines how the signals ALUSrc and RegDst are set.
    - Furthermore, an R-type instruction writes a register (Reg-Write = 1), but neither reads nor writes data memory.
    - When the Branch control signal is 0, the PC is unconditionally replaced with PC + 4; otherwise, the PC is replaced by the branch target if the Zero output of the ALU is also high.
    - The ALUOp field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the funct field.
- The second and third rows of this table give the control signal settings for lw and sw.
    - These ALUSrc and ALUOp fields are set to perform the address calculation.
    - The MemRead and MemWrite are set to perform the memory access. Finally, RegDst and RegWrite are set for a load to cause the result to be stored into the rt register.

- The branch instruction is similar to an R-format operation, since it sends the rs and rt registers to the ALU.
- The ALUOp field for branch is set for a subtract (ALU control = 01), which is used to test for equality.
- Notice that the MemtoReg field is irrelevant when the RegWrite signal is 0: since the register is not being written, the value of the data on the register data write port is not used.
- Thus, the entry MemtoReg in the last two rows of the table is replaced with X for don't care.
- Don't cares can also be added to RegDst when RegWrite is 0. This type of don't care must be added by the designer, since it depends on knowledge of how the datapath works.

R-Type Instruction



Chapter 4 — The Processor — 28

- The datapath in operation for an R-type instruction, such as add $t1,$t2,$t3. The control lines, datapath units, and connections that are active are highlighted.

- Although everything occurs in one clock cycle, we can think of four steps to execute the instruction; these steps are ordered by the flow of information:

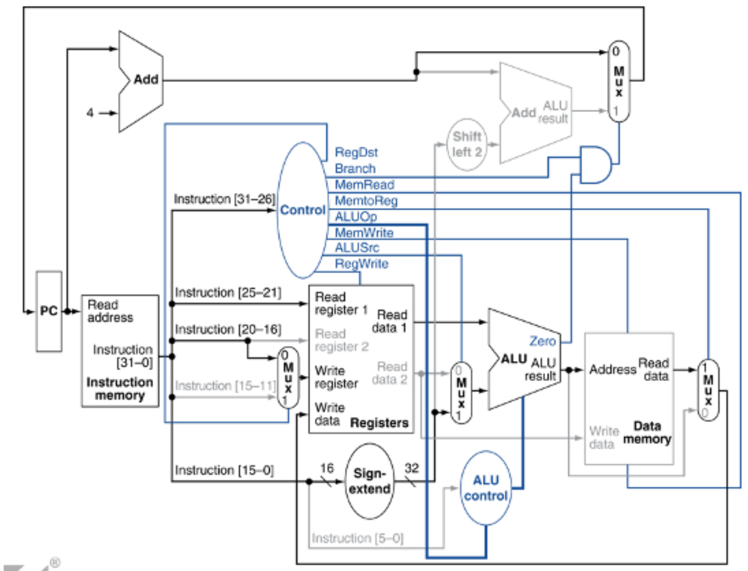
1. The instruction is fetched, and the PC is incremented.

2. Two registers, $t2 and $t3, are read from the register file; also, the main control unit computes the setting of the control lines during this step.

3. The ALU operates on the data read from the register file, using the function code (bits 5:0, which is the funct field, of the instruction) to generate the ALU function.

4. The result from the ALU is written into the register file using bits 15:11 of the instruction to select the destination register ($t1).

Load Instruction



Chapter 4 — The Processor — 29

- The datapath in operation for a load instruction. The control lines, datapath units, and connections that are active are highlighted.
- A store instruction would operate very similarly. The main difference would be that the memory control would indicate a write rather than a read, the second register value read would be used for the data to store, and the operation of writing the data memory value to the register file would not occur.
- We can think of a load instruction as operating in five steps (similar to how the R-type executed in four):
- For example, lw $t1, offset($t2)

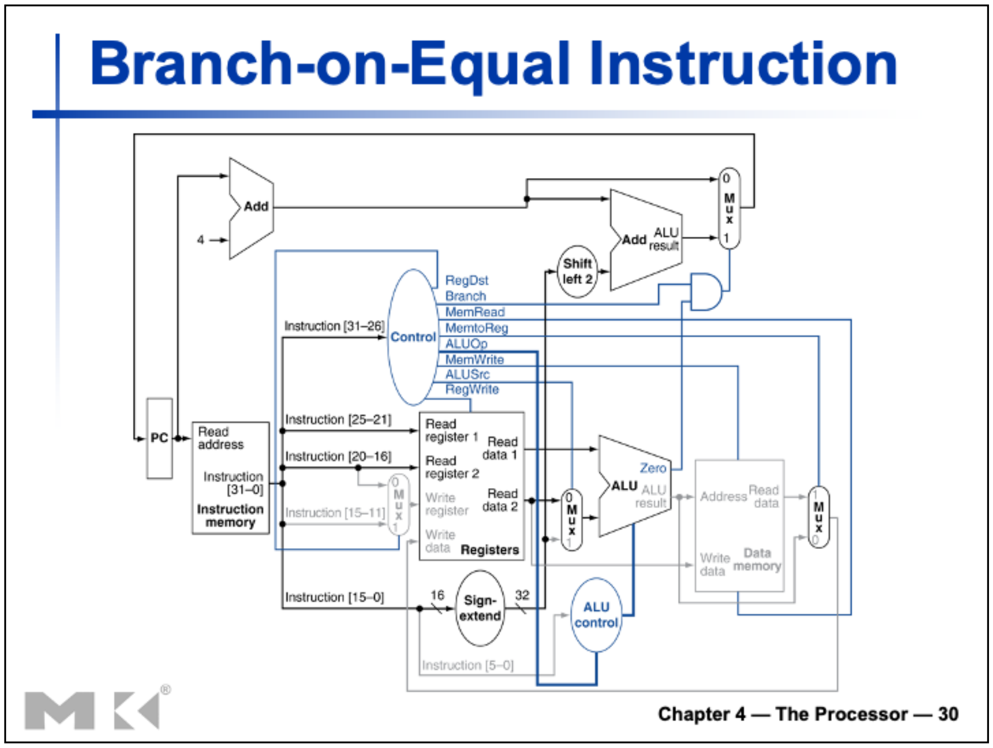
1. An instruction is fetched from the instruction memory, and the PC is incremented.

2. A register ($t2) value is read from the register file.

3. The ALU computes the sum of the value read from the register file and the sign-extended, lower 16 bits of the instruction (offset).

4. The sum from the ALU is used as the address for the data memory.

5. The data from the memory unit is written into the register file; the register destination is given by bits 20:16 of the instruction ($t1).

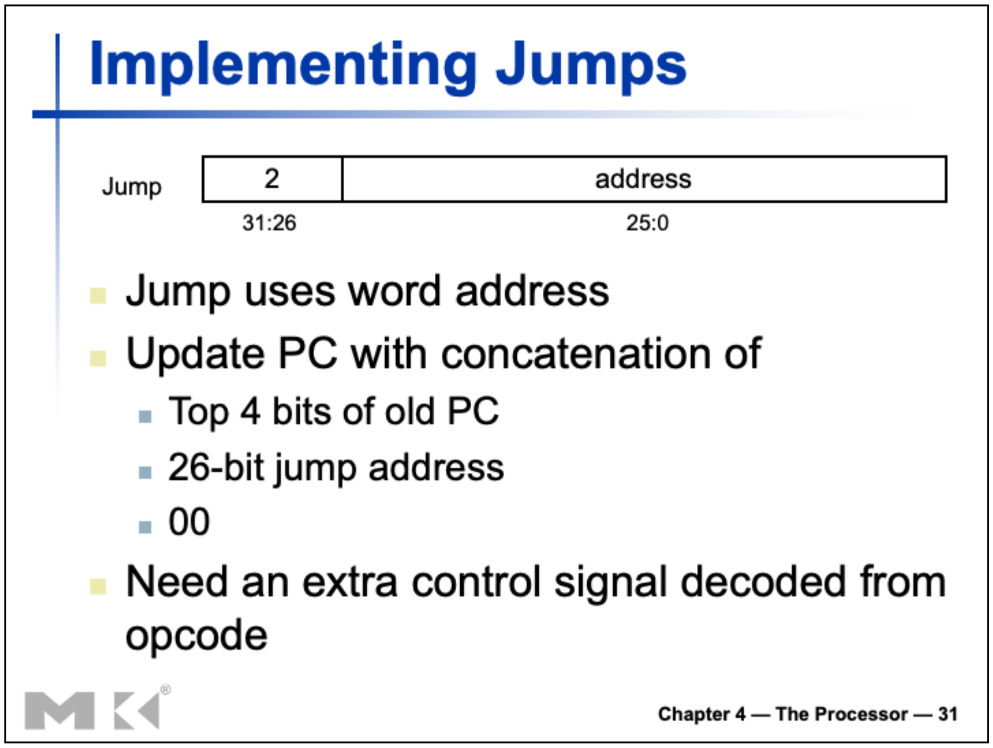


Branch-on-Equal Instruction

Chapter 4 — The Processor — 30

- Finally, we can show the operation of the branch-on-equal instruction, such as beq $t1,$t2,offset, in the same fashion.
- It operates much like an R-format instruction, but the ALU output is used to determine whether the PC is written with PC + 4 or the branch target address.
- the four steps in execution:
  - An instruction is fetched from the instruction memory, and the PC is incremented.
  - Two registers, $t1 and $t2, are read from the register file.
  - The ALU performs a subtract on the data values read from the register file. The value of PC + 4 is added to the sign-extended, lower 16 bits of the instruction (offset) shifted left by two; the result is the branch target address.
  - The Zero result from the ALU is used to decide which adder result to store into the PC.

- Figure shows the datapath in operation for a branch-on-equal instruction. The control lines, datapath units, and connections that are active are highlighted.
- After using the register file and ALU to perform the compare, the Zero output is used to select the next program counter from between the two candidates.



Implementing Jumps

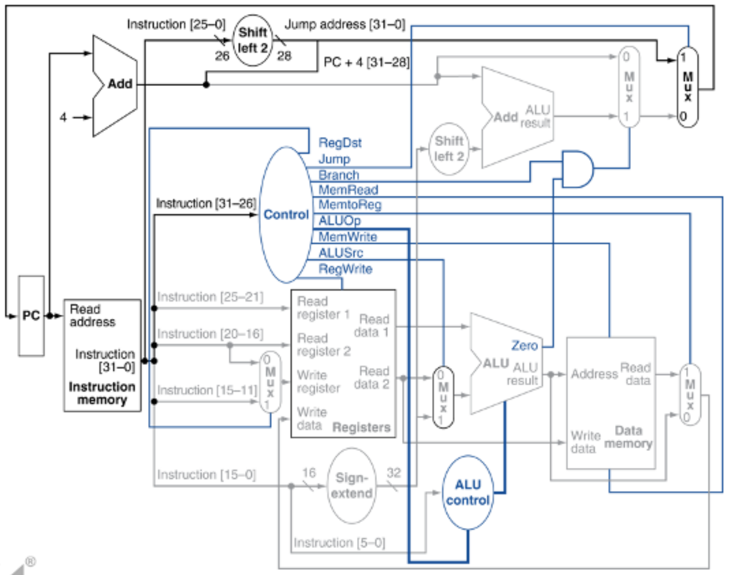
Jump	2	address
	31:26	25:0

- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00
- Need an extra control signal decoded from opcode

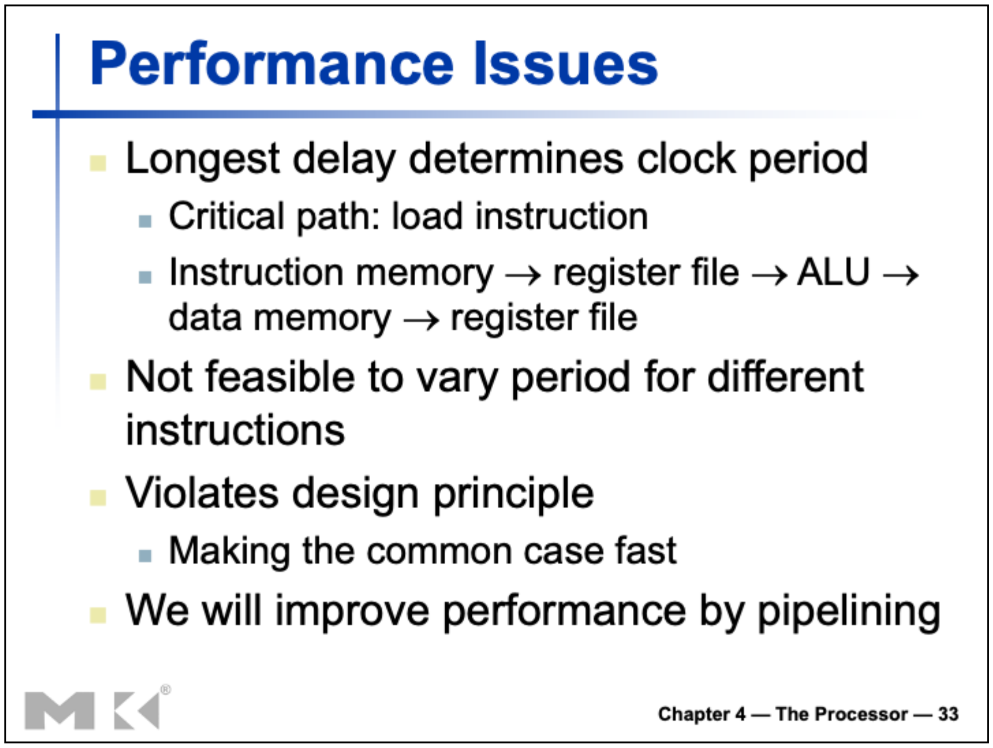
Chapter 4 — The Processor — 31

- The jump instruction looks somewhat like a branch instruction but computes the target PC differently and is not conditional.
- Like a branch, the low-order 2 bits of a jump address are always 00 base two.
- The next lower 26 bits of this 32-bit address come from the 26-bit immediate field in the instruction.
- The upper 4 bits of the address that should replace the PC come from the PC of the jump instruction plus 4.
- Thus, we can implement a jump by storing into the PC the concatenation of
  - the upper 4 bits of the current PC + 4 (these are bits 31:28 of the sequentially following instruction address)
  - the 26-bit immediate field of the jump instruction
  - the bits 00 base two

Datapath With Jumps Added



- The simple control and datapath are extended to handle the jump instruction.
- An additional multiplexor (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one. This multiplexor is controlled by the jump control signal.
- The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address.



Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
- Violates design principle
  - Making the common case fast
- We will improve performance by pipelining

Chapter 4 — The Processor — 33

- Why a Single-Cycle Implementation Is Not Used Today? – because it is inefficient
  - To see why this is so, notice that the clock cycle must have the same length for every instruction in this single-cycle design.
  - Of course, the longest possible path in the processor determines the clock cycle.
  - This path is almost certainly a load instruction, which uses five functional units in series: the instruction memory, the register file, the ALU, the data memory, and the register file.
  - Although the CPI is 1 (see Chapter 1), the overall performance of a single-cycle implementation is likely to be poor, since the clock cycle is too long.

- The penalty for using the single-cycle design with a fixed clock cycle is significant, but might be considered acceptable for this small instruction set.
- Historically, early omputers with very simple instruction sets did use this implementation technique. However, if we tried to implement the floating-point unit or an instruction set with more complex instructions, this single-cycle design wouldn't work well at all.
- Because we must assume that the clock cycle is equal to the worst-case delay for all instructions, it's useless to try implementation techniques that reduce the delay of the common case but do not improve the worst-case cycle time.

- In next section, we’ll look at another implementation technique, called pipelining, that uses a datapath very similar to the single-cycle datapath but is much more efficient by having a much higher throughput. Pipelining improves efficiency by executing multiple instructions simultaneously.