

CHAPTER 4

The Processor

Chia-Heng Tu
Dept. of Computer Science and Information Engineering
National Cheng Kung University

國立成功大學
National Cheng Kung University

Instruction Execution Overview

- 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 lw 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
  - ➢ The actions are largely the same for the three instruction classes (memory-reference, arithmetic-logical, and branches)
  - ➢ Simplicity and regularity of RISC-V instruction set simplify the implementation

Instruction Execution Overview (Cont'd)

3. All instruction classes use the arithmetic-logical unit (ALU) after reading the registers
   - memory-reference inst. use the ALU for an address calculation
   - the arithmetic-logical inst. for the operation execution, and
   - conditional branches for the equality test

4. 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 or load instruction must write the data from the ALU or memory back into a register
   - A conditional branch instruction may need to change the next instruction address (PC → target address) based on the comparison; otherwise, the PC should be incremented by four (PC → PC + 4) to get the address of the subsequent instruction

Instruction Execution Flow

1. Read from PC
2. Read from Reg.
3. Use ALU
   - Arithmetic operation
   - Memory addr. Calculation
   - Branch condition determination

4. Actions for each class differ
   - (Mem.) Load/Store data
   - (Arith.) Write ALU result to reg
   - (Cond.) PC ← target addr. (may be = PC+4)

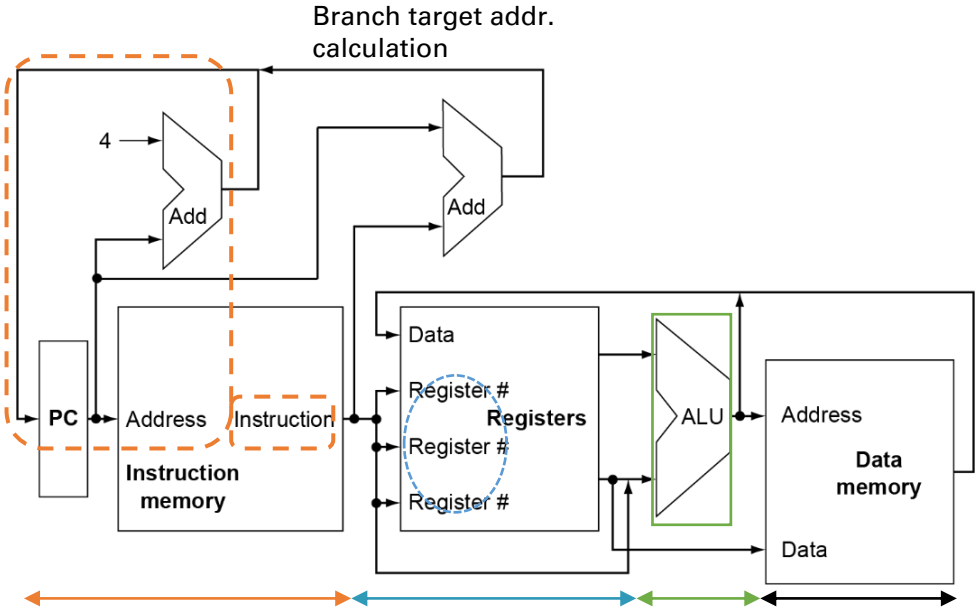


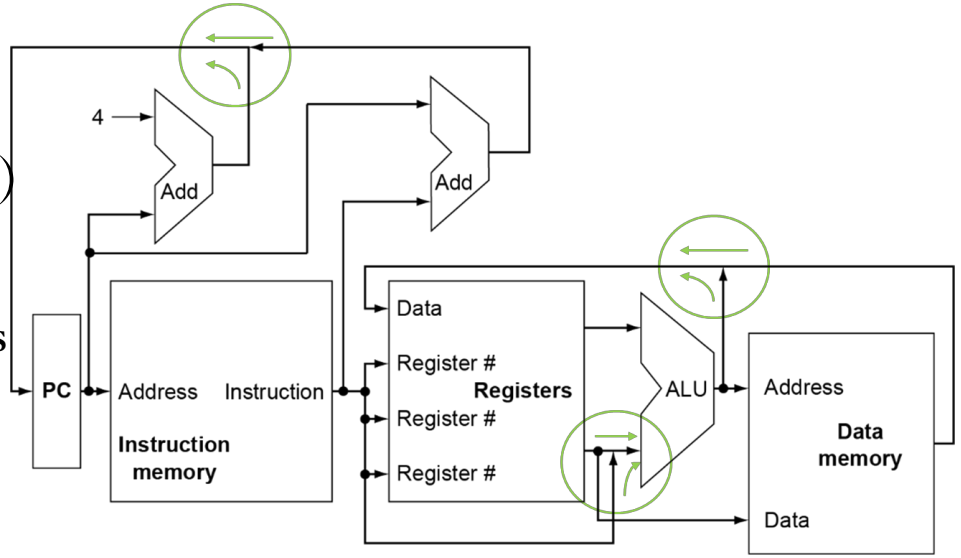
FIGURE 4.1 An abstract view of the implementation of the RISC-V subset showing the major functional units and the major connections between them.
All instructions start by using the program counter to supply the instruction address to the instruction memory. After the instruction is fetched, the register 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 an equality check (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 load a value from memory into the registers or store a value from the registers. The result from the ALU or memory is written back into the register file. Branches require the use of the ALU output to determine the next instruction address, which comes either from the adder (where the PC and branch offset are summed) or from an adder that increments the current PC by four.
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

Instruction Execution Flow (Cont'd)

- Fig. 4.1 focuses on functional units and their interconnection

- It omits two important aspects:

1. Three logic elements (multiplexors) should be added
   - ➢ to choose from among the multiple sources and steers one of those sources to its destination; also known as data selector
   - ➢ Fig. 4.2 gives the details for the three Mux
2. A control unit should be added …



A data selector selects from among several inputs based on the setting of its control lines; The control lines are set according to information taken from the instruction being executed

Instruction Execution Flow (Cont'd)

- It omits two important aspects:
2. A control unit should be added
  - ➢to determine how to set the control lines for the functional units and two of the multiplexors
  - ➢based on various fields in the instruction (being executed) direct these operations
- Control examples:
  - ➢The data memory must read on a load and write on a store
  - ➢The register file must be written only on a load or an arithmetic-logical instruction
  - ➢The ALU must perform one of several operations
- Fig. 4.2 adds the multiplexors and control unit on Fig. 4.1

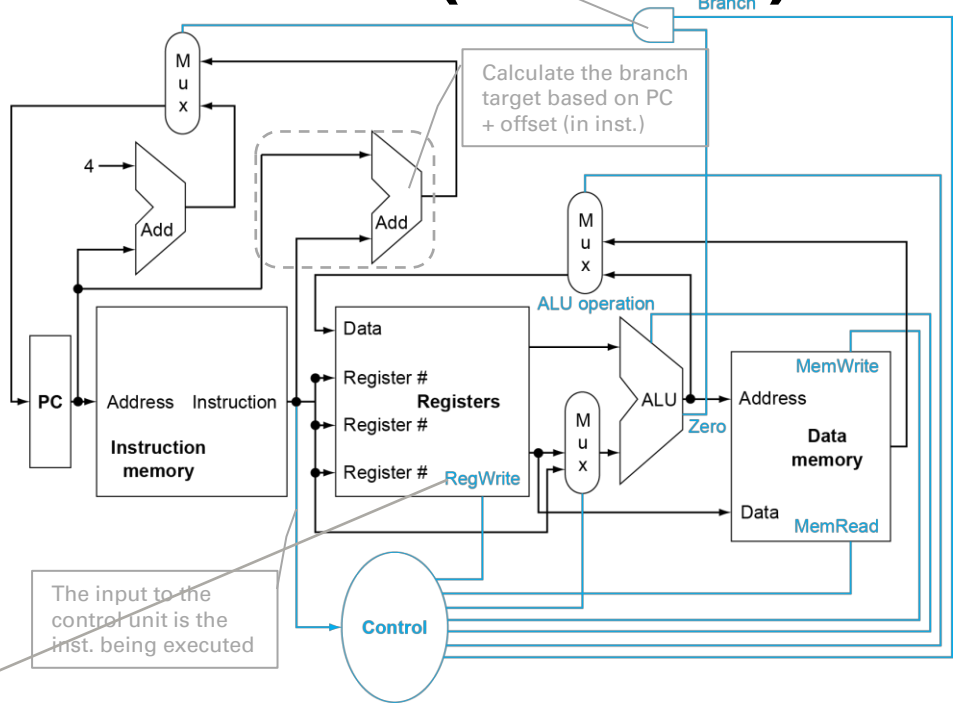


FIGURE 4.2 The basic implementation of the RISC-V subset, including the necessary multiplexors and control lines.

1. The top multiplexor ("Mux") controls what value replaces the PC (PC + 4 or the branch destination address); 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.
2. 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.
3. The bottom-most multiplexor is used to determine whether the second ALU input is from the registers (for an arithmetic-logical instruction or a branch) 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

A Simplified (Single Cycle) Version

- Logical designs in a processor (Sec. 4.2)
- Datapath elements of the processor (Sec. 4.3)
- Control for the single cycle processor (Sec. 4.4)

Logic Design Basics

- Review a few key ideas in digital logic that we will use extensively
  - If you are unfamiliar with these, you can refer to Appendix A

- Two types of the datapath elements in RISC-V

1. Elements that operate on data values
   - A combinational element is an operational element, such as an AND gate or an ALU, to process data
   - Its outputs depend only on the current inputs
     - Given the same input, a combinational element always produces the same output
2. Elements that contain state
   - A state element is a memory element, such as a register or a memory
   - An element contains state if it has some internal (nonvolatile) storage
   - Output is related to input & current states

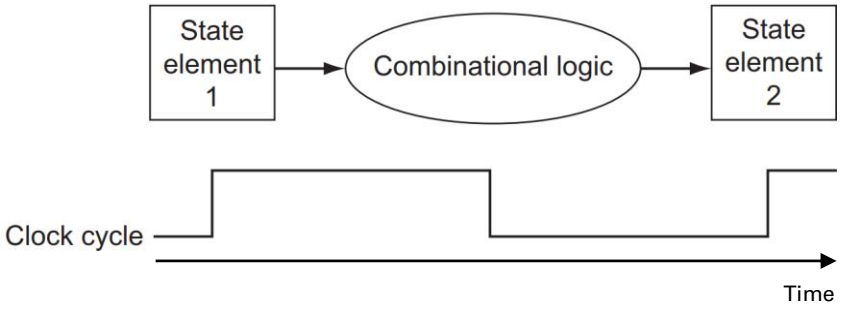


FIGURE 4.3 Combinational logic, state elements, and the clock are closely related. In a synchronous digital system, the clock determines when elements with state will write values into internal storage. Any inputs to a state element must reach a stable value (that is, have reached a value from which they will not change until after the clock edge) before the active clock edge causes the state to be updated. All state elements in this chapter, including memory, are assumed positive edge-triggered; that is, they change on the rising clock edge

- A state element has at least 2 inputs and 1 output
- The required inputs are the data value to be written into the element and the clock, which determines when the data value is written
- The output from a state element provides the value that was written in an earlier clock cycle

Logic Design Basics (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 that 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 is assumed
  - It is a clocking scheme in which all state changes occur on a clock edge
  - Any values stored in a sequential logic element are updated only on a clock edge, which is a quick transition from low to high or vice versa
  - The inputs are values that were written in a previous clock cycle, while the outputs are values that can be used in a following clock cycle
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses

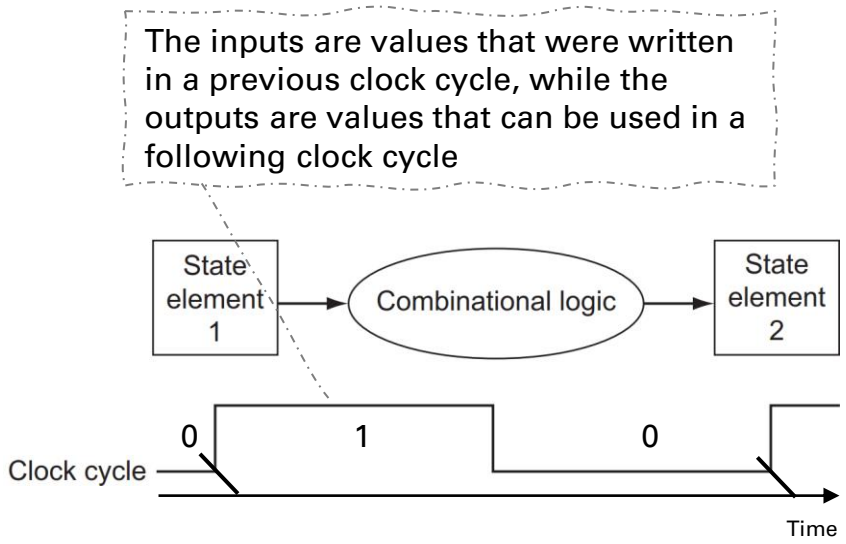
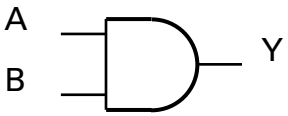


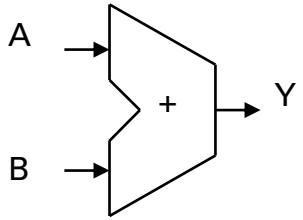
FIGURE 4.3 Combinational logic, state elements, and the clock are closely related. In a synchronous digital system, the clock determines when elements with state will write values into internal storage. Any inputs to a state element must reach a stable value (that is, have reached a value from which they will not change until after the clock edge) before the active clock edge causes the state to be updated. All state elements in this chapter, including memory, are assumed positive edge-triggered; that is, they change on the rising clock edge

Recap: Combinational Elements

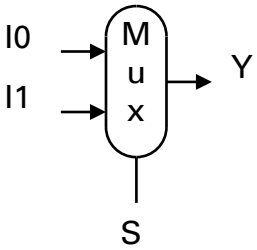
- AND
  - Y = A & B



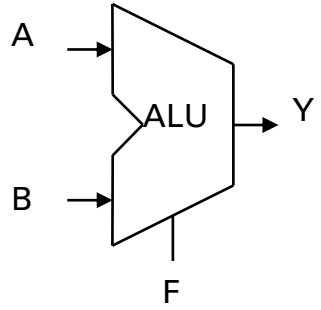
- Adder
  - Y = A + B



- Multiplexer
  - Y = S ? I1 : I0

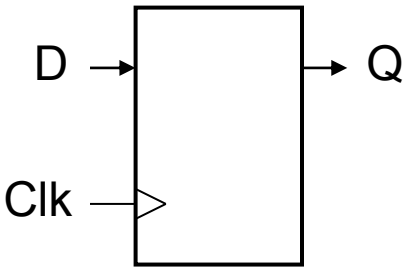


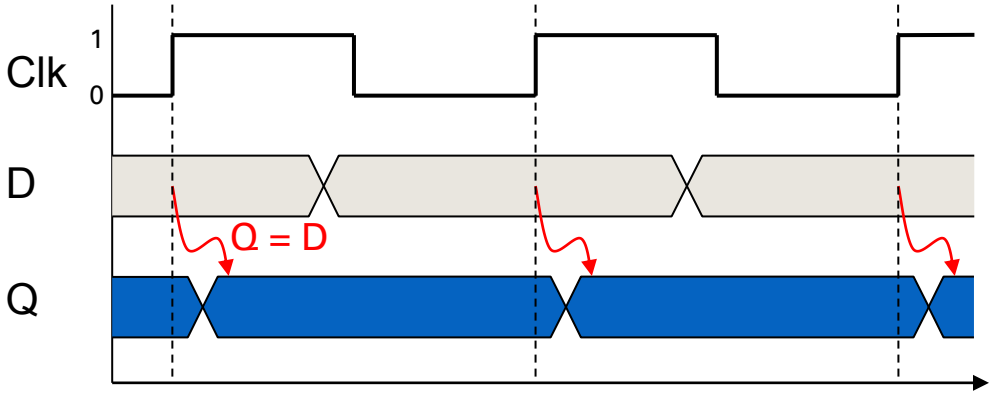
- Arithmetic/Logic Unit
  - Y = F(A, B)



Recap: Sequential Elements

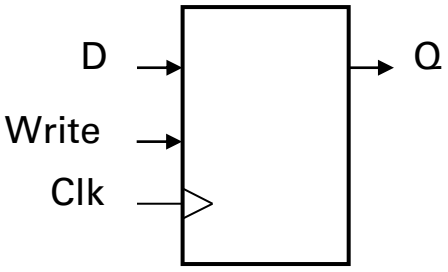
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes (0 → 1, or 1 → 0)
  - A positive edge-triggered example below; clock changes 0 → 1

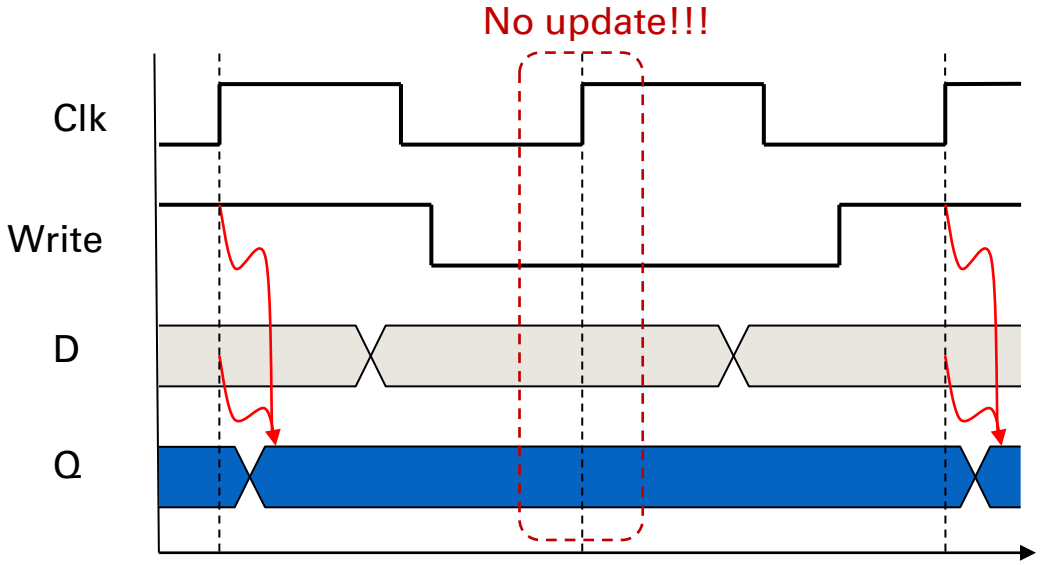




Recap: Sequential Elements w/ Write Enable

- Register with write control
  - Only updates on clock edge when Write control input is 1
  - Used when stored value is required later
  - Q is changed when edge changes and Write == 1





Datapath Elements (Inst. Fetch)

- A datapath element
  - A unit used to operate on or hold data within a processor
  - In the RISC-V implementation, the datapath elements include the instruction and data memories, the register file, the ALU, and adders
- We will build a RISC-V datapath incrementally

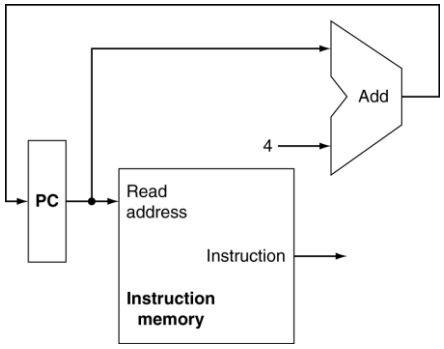


FIGURE 4.6 A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched inst. is used by other parts of the datapath

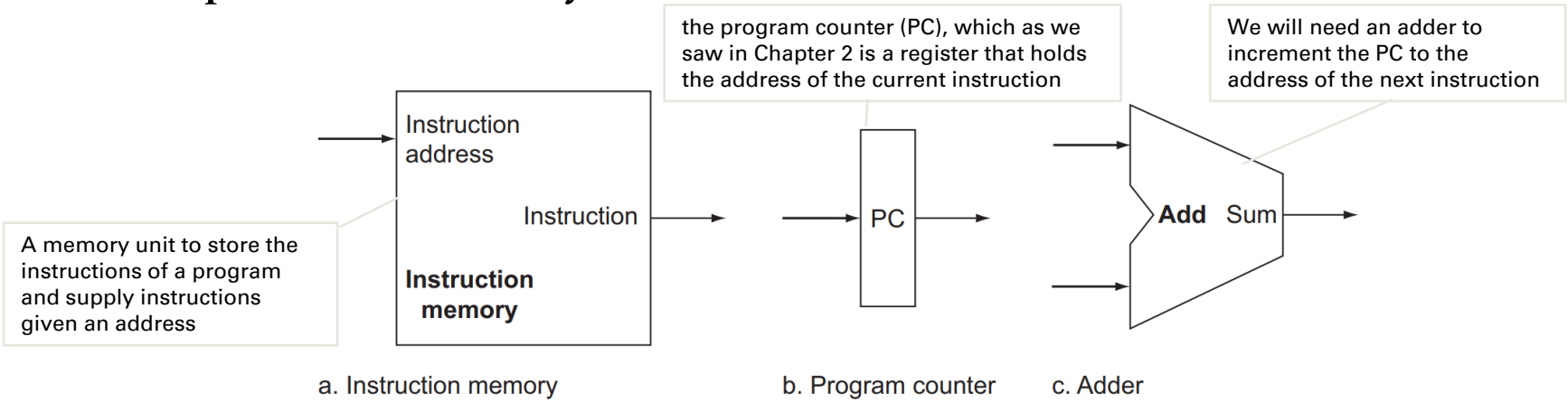


FIGURE 4.5 Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address. The state elements are the instruction memory and the program counter. The instruction memory need only provide read access because the datapath does not write instructions. 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

Datapath Elements (R-format ALU Operations)

- A register file
  - ➢is a structure to keep the collection of 32 general-purpose registers
  - ➢in which any register can be read or written by specifying the number of the register in the file
  - ➢contains the register state of the computer

A total of three inputs (two 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 (2^5)

- Arithmetic inst.
  - ➢ E.g., add x1, x2, x3
- Load/Store inst.
  - ➢ lw x1, offset(x2) or
  - ➢ sw x1, offset(x2)

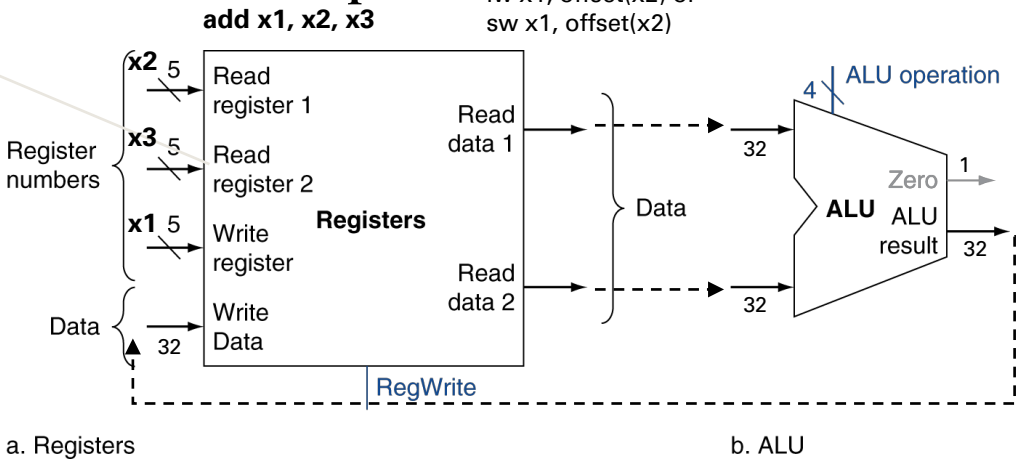
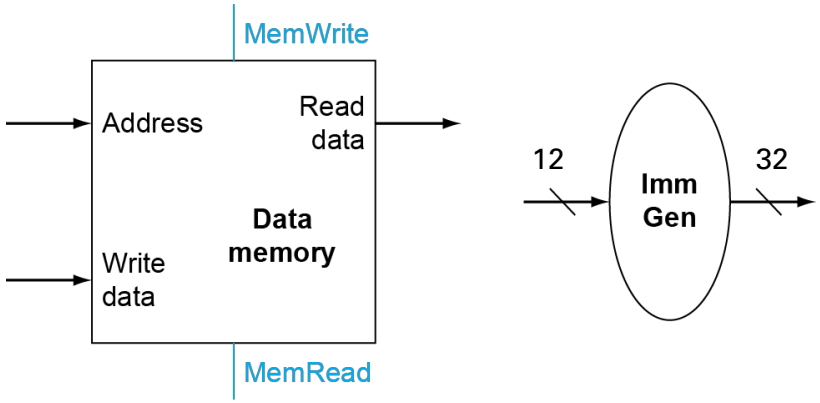


FIGURE 4.7 The two elements needed to implement R-format ALU operations are the register file and the ALU. The register file contains all the registers and has two read ports and one write port. The design of multiported register files is discussed in Section A.8 of Appendix A. The register file always 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. The inputs carrying the register number to the register file are all 5 bits wide, whereas the lines carrying data values are 32 bits wide. The operation to be performed by the ALU is controlled with the ALU operation signal, which will be 4 bits wide, using the ALU designed in Appendix A. We will use the Zero detection output of the ALU shortly to implement conditional branches.

Datapath Elements (Load/Store)

- To further support Load/Store inst., it needs
  - a unit to sign-extend the 12-bit offset field in the instruction to a 32-bit signed value, and
  - a data memory unit to read from (load) or write to (store) registers

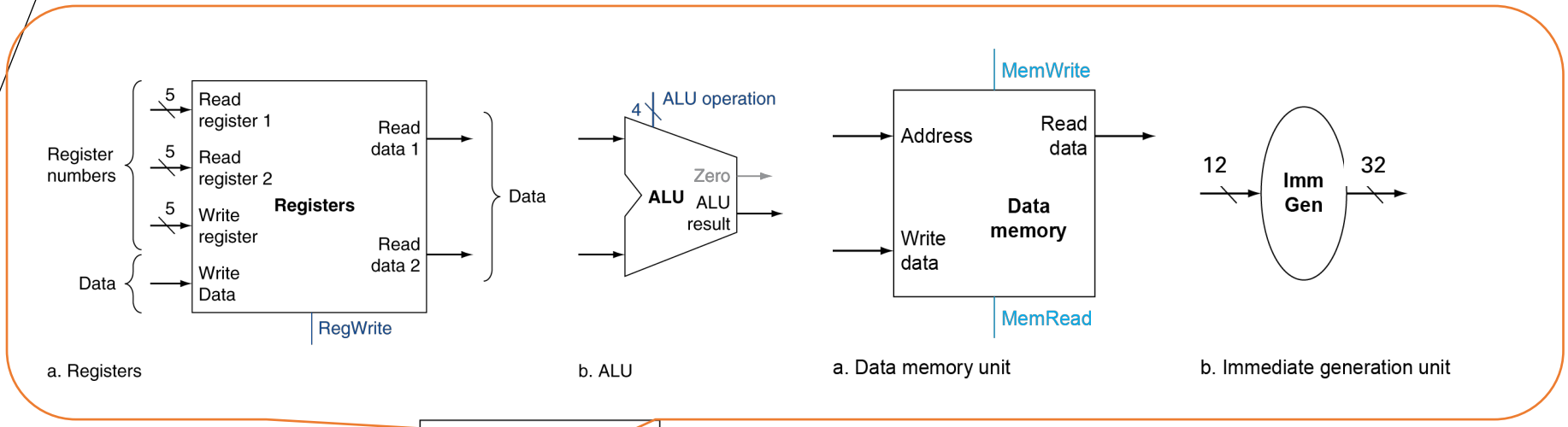
- Arithmetic inst.
  - E.g., add x1, x2, x3
- Load/Store inst.
  - lw x1, offset(x2) // load word from mem.
  - sw x1, offset(x2) // store word to mem.



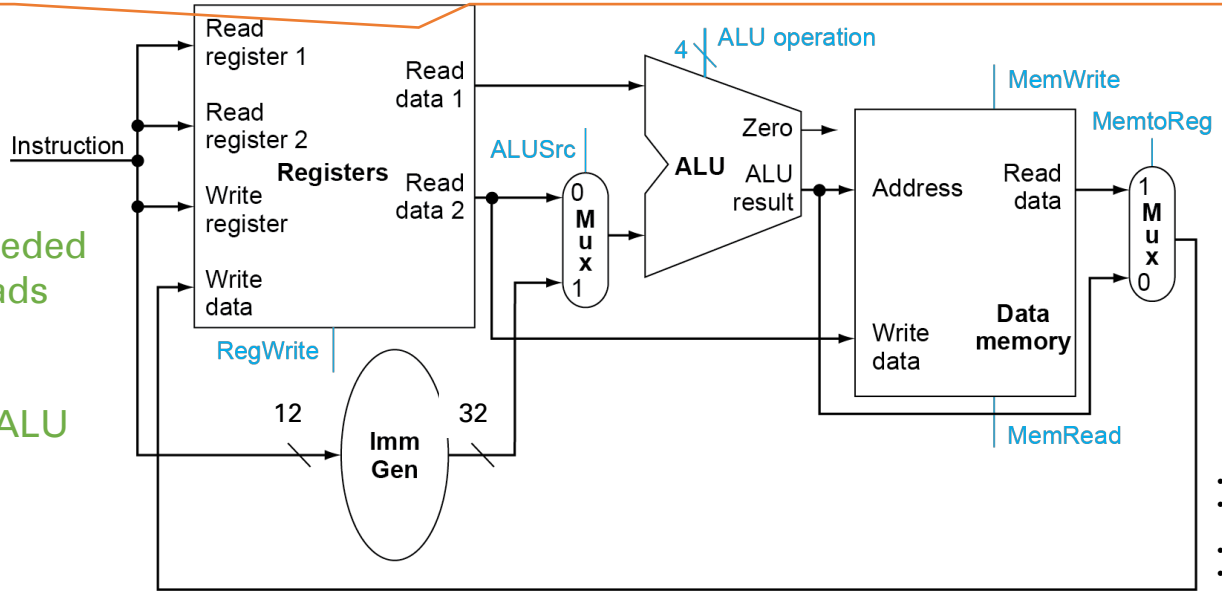
a. Data memory unit b. Immediate generation unit

FIGURE 4.8 The two units needed to implement loads and stores, in addition to the register file and ALU of Figure 4.7, are the data memory unit and the immediate generation unit. The memory unit is a state element with inputs for the address and the write data, and a single output for the read result. There are separate read and write controls, although only one of these may be asserted on any given clock. The memory unit needs a read signal, since, unlike the register file, reading the value of an invalid address can cause problems, as we will see in Chapter 5. The immediate generation unit (ImmGen) has a 32-bit instruction as input that selects a 12-bit field for load, store, and branch if equal that is sign-extended into a 32-bit result appearing on the output (see Chapter 2). We assume the data memory is edge-triggered for writes. Standard memory chips actually have a write enable signal that is used for writes. Although the write enable is not edge-triggered, our edge-triggered design could easily be adapted to work with real memory chips. See Section A.8 of Appendix A for further discussion of how real memory chips work

Datapath for Load/Store Inst.



The two units needed to implement loads and stores, in addition to the register file and ALU of Figure 4.7.

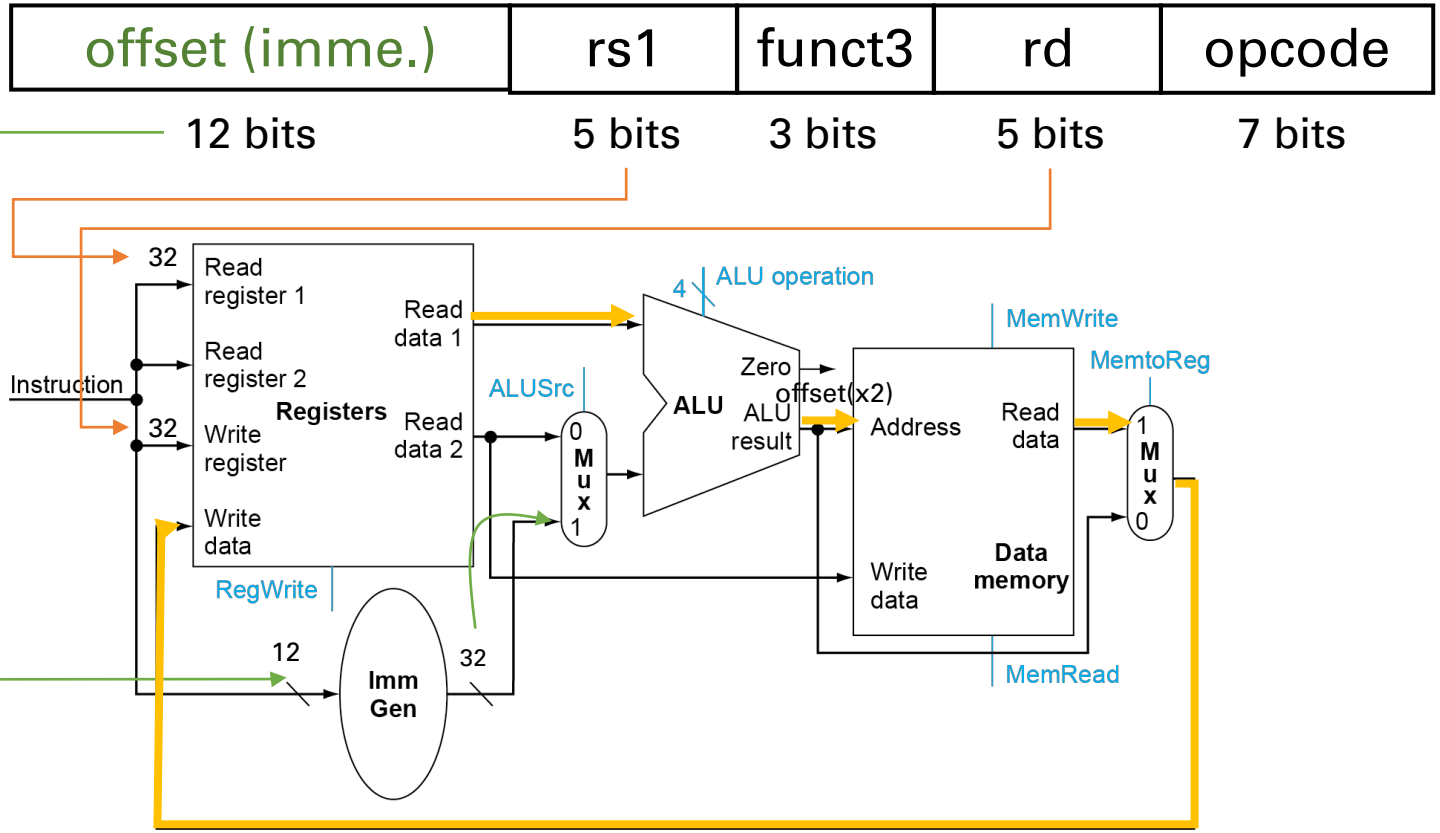


- Read register operands
- Calculate address using 12-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update reg.
- Store: Write register value to memory

FIGURE 4.10 The datapath for the memory instructions and the R-type instructions. This example shows how a single datapath can be assembled from the pieces in Figures 4.7 and 4.8 by adding multiplexors. Two multiplexors are needed, as described in the example

Data Flow for Load Inst.

- I-format inst.
  - lw x1, offset(x2)
- You can do the same for store inst.



Recap: Branch Inst.

- SB format for conditional branches
  - A 7-bit opcode, a 3-bit function code, two 5-bit register operands (rs1 and rs2), and a 12-bit address immediate
  - bne x10, x11, 2000_{10}
  - If x10 != x11, go to location PC + 2000_{10} (= 0111 1101 0000)

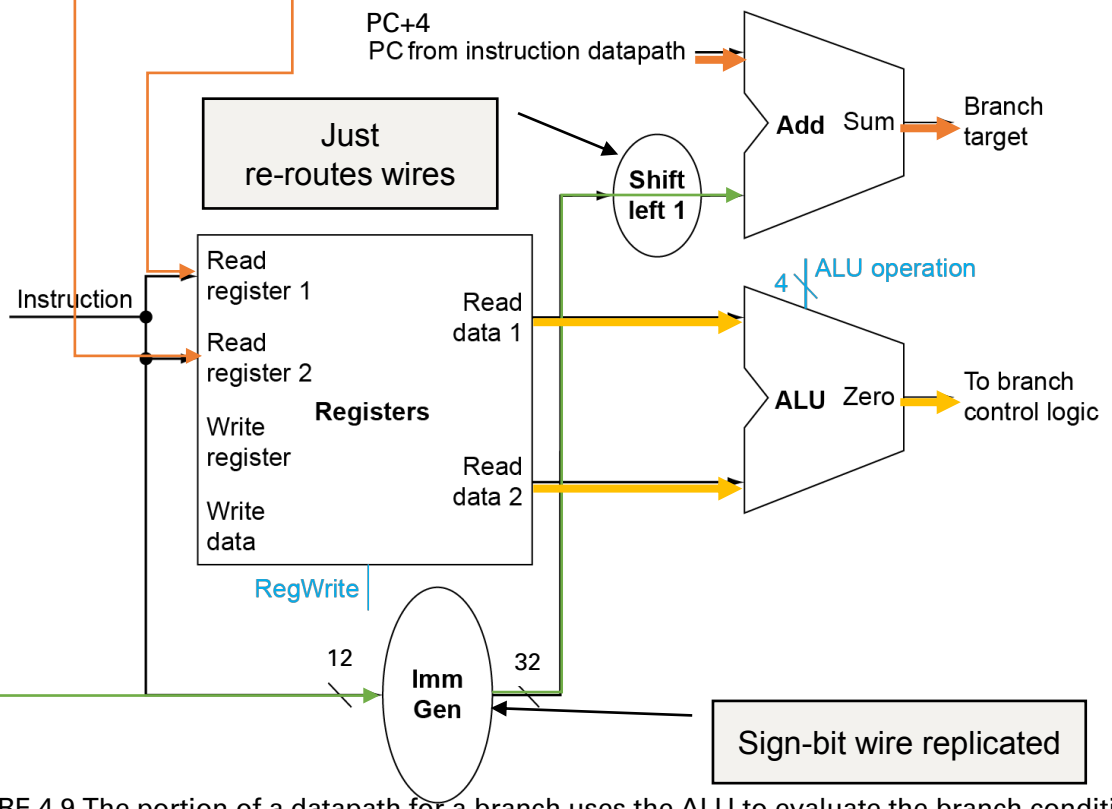
0011111	01011	01010	001	01000	1100111
imm[12:6]	rs2	rs1	funct3	imm[5:1]	opcode

- The operations done for a branch inst.
  - Read register operands
  - Compare operands
    - Use ALU, subtract and check Zero output
  - Calculate target address
    - Sign-extend displacement
    - Shift left 1 place (halfword displacement)
    - Add to PC+4 value
      when executing the branch inst. (at the instruction fetch stage), PC is updated to PC+4

Data Flow for Branch Inst.

bne x1, x2, 2000_{10}
//If x1 != x2, go to location (PC+4) + 2000_{10} (= 0111 1101 0000)

0011111	01011	01010	001	01000	1100111
imm[12:6]	rs2	rs1	funct3	imm[5:1]	opcode



- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 1 place (halfword displacement)
  - Add to PC+4 value when executing the branch inst. (at the instruction fetch stage), PC is updated to PC+4

FIGURE 4.9 The portion of a datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the PC and immediate (the branch displacement). Control logic is used to decide whether the incremented PC or branch target should replace the PC, based on the Zero output of the ALU

Compose the Elements

- Let datapath do an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - It needs to separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

Datapath of R-format, Load, Store Inst.

- 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 12-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), which is determined by the Mux

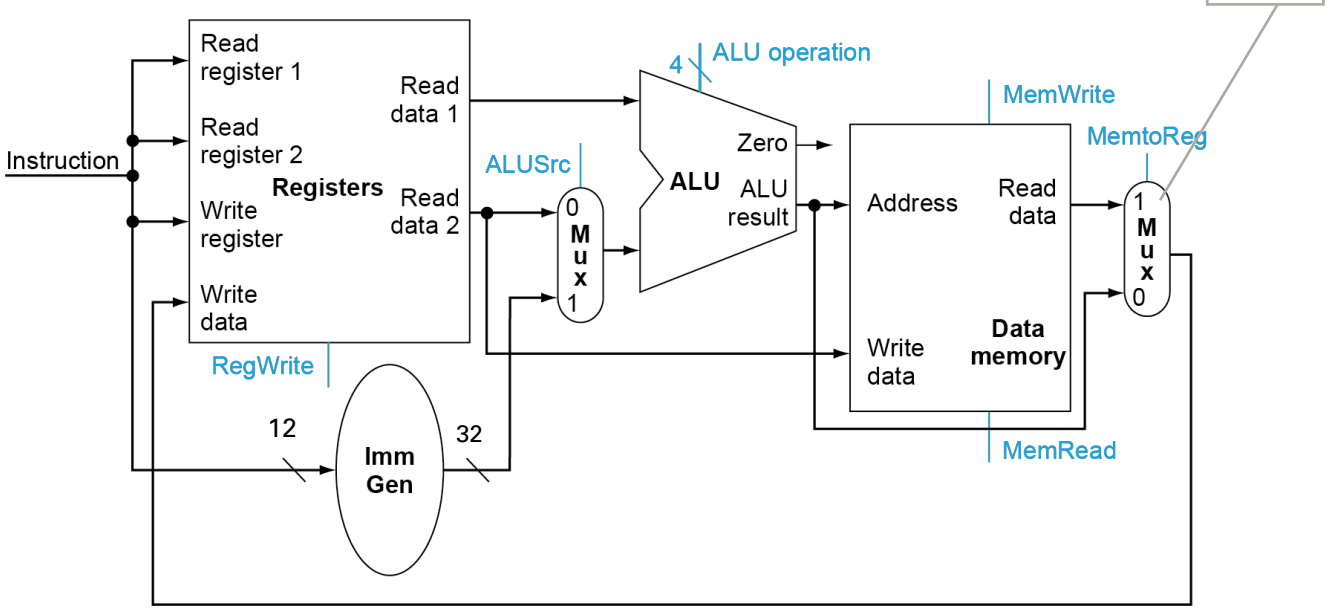


FIGURE 4.10 The datapath for the memory instructions and the R-type instructions. This example shows how a single datapath can be assembled from the pieces in Figures 4.7 and 4.8 by adding multiplexors. Two multiplexors are needed, as described in the example

Simple Datapath for Core RISC-V Arch.

- Integrated with the above data elements
- Runs the instructions in a single clock cycle
  - I.e., load-store register, ALU operations, and branches

You should be able to identify the "routes" for arithmetic, memory, and branch instructions on the datapath

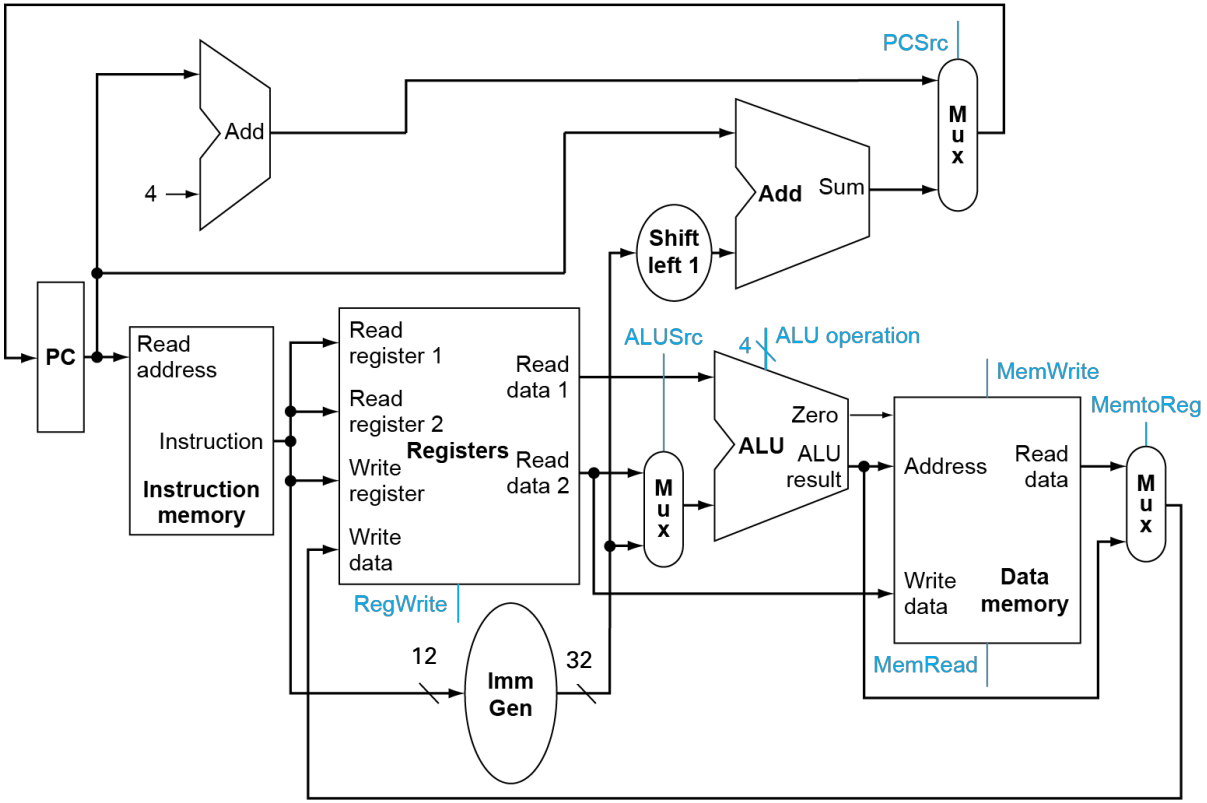
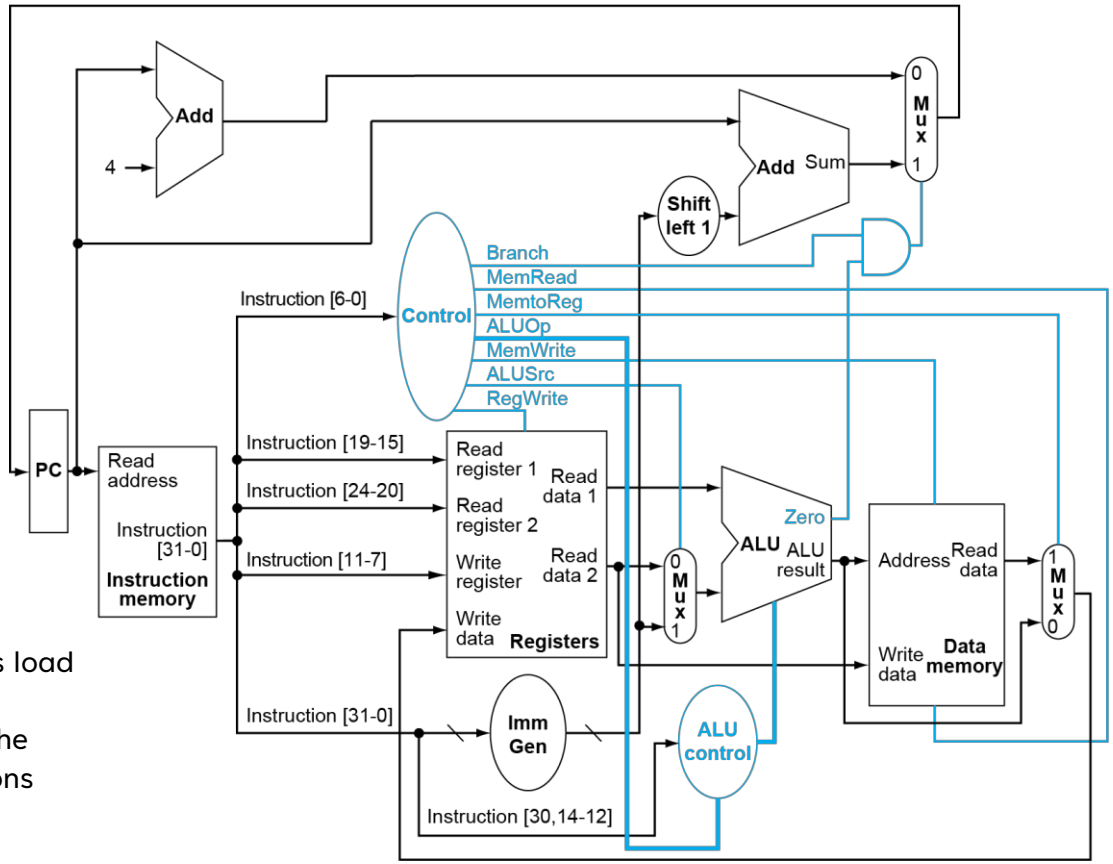


FIGURE 4.11 The simple datapath for the core RISC-V architecture combines the elements required by different instruction classes. The components come from Figures 4.6, 4.9, and 4.10. This datapath can execute the basic instructions (load-store register, ALU operations, and branches) in a single clock cycle. Just one additional multiplexor is needed to integrate branches

The Control of the Processor

To add the *control* on the built datapath



This simple datapath covers load word (lw), store word (sw), branch if equal (beq), and the arithmetic-logical instructions add, sub, and, and or

FIGURE 4.21 The simple datapath with the control unit. The input to the control unit is the 7-bit opcode field from the instruction. The outputs of the control unit consist of two 1-bit signals that are used to control multiplexors (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. Notice that PCSrc is now a derived signal, rather than one coming directly from the control unit. Thus, we drop the signal name in subsequent figures

The ALU Control (Operations in ALU)

- Four possible functions could be performed by the ALU
  - ➢AND, OR, add, subtract
  - ➢depending on the value of
    - ❖the 7-bit funct7 field (bits 31:25) and
    - ❖the 3-bit funct3 field (bits 14:12)

- ALU used for
  - ➢Load/Store inst. by addition
  - ➢Branch inst. by subtraction
  - ➢R-type inst. depending on opcode

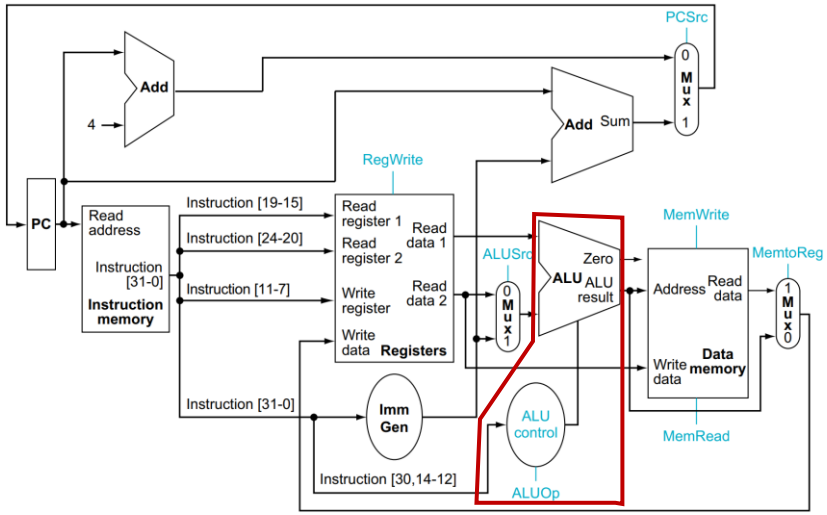


FIGURE 4.19 The datapath of Figure 4.11 with all necessary multiplexors and all control lines identified. The control lines are shown in color. The ALU control block has also been added, which depends on the funct3 field and part of the funct7 field. The PC does not require a write control, since it is written once at the end of every clock cycle; the branch control logic determines whether it is written with the incremented PC or the branch target address

The ALU Control (Inputs and Output)

ALU control

- has two inputs
  1. ALUOp: a 2-bit control field
  2. inst. ops (funct7 and funct3 fields)
- has an output
  3. the 4-bit ALU *control* input

- ALUOp value depends on the input inst., and it performs
  - ➢add (00) for loads and stores,
  - ➢subtract and test if zero (01) for beq, or
  - ➢? (10) further determined by the operation encoded in the funct7 and funct3 fields
  - ➢See Fig. 4.12

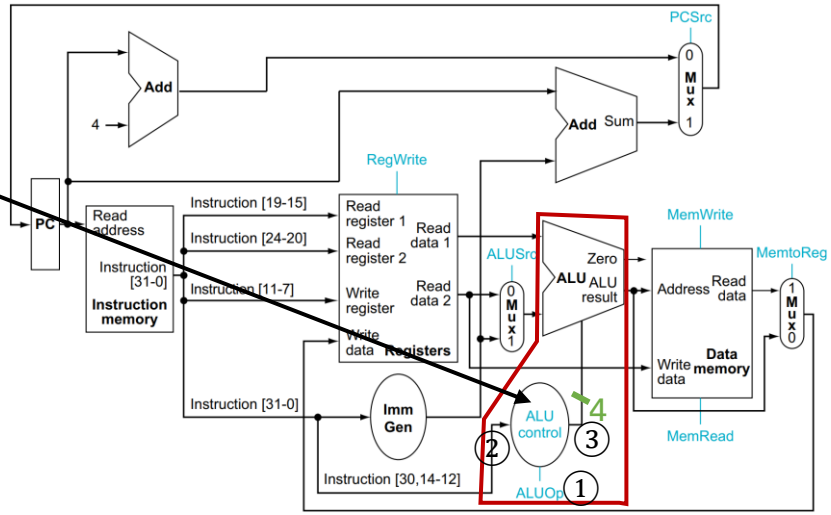


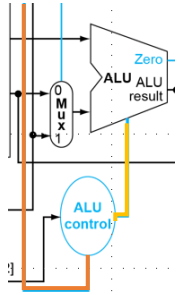
FIGURE 4.19 The datapath of Figure 4.11 with all necessary multiplexors and all control lines identified. The control lines are shown in color. The ALU control block has also been added, which depends on the funct3 field and part of the funct7 field. The PC does not require a write control, since it is written once at the end of every clock cycle; the branch control logic determines whether it is written with the incremented PC or the branch target address

Name (Bit position)	Fields 31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

The ALU Control (Mapping Inputs to Output Signals)

- Inputs and output of the ALU control
- Multiple levels of input decoding (ALUOp, Funct7, Funct3 fields) may help reduce the latency of the control unit → a critical factor in determining the clock cycle time

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract



Inputs

Output

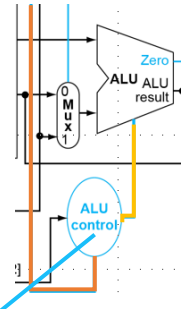
Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
lw	00	load word	XXXXXXX	XXX	add	0010
sw	00	store word	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

- ALUOp value depends on the input inst., and it performs
  - add (00) for loads and stores,
  - subtract and test if zero (01) for beq, or
  - ? (10) further determined by the operation encoded in the funct7 and funct3 fields
  - See Fig. 4.12

FIGURE 4.12 How the ALU control bits are set depends on the ALUOp control bits and the different opcodes for the R-type instruction. The instruction, 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 funct7 or funct3 fields; in this case, we say that we "don't care" about the value of the opcode, and the bits are shown as Xs. When the ALUOp value is 10, then the funct7 and funct3 fields are used to set the ALU control input. See Appendix A

The ALU Control (The Truth Table)

- The truth table below shows how the 4-bit ALU control is set depending on these input fields
  - ➢ It shows only the truth table entries for which the ALU control must have a specific value
  - ➢ A don't-care term
    - ❖ represented by an X in an input column
    - ❖ indicates that the output does not depend on the value of the input corresponding to that column



- If you are interested in the *logical design* of an ALU
  - ➢ refer to Fig. A.5.8 of Appendix A (Sec. A.5) provides the details of using logical units to build a 1-bit ALU

ALUOp		Funct7 field							Funct3 field			
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

FIGURE 4.13 The truth table for the 4 ALU control bits (called Operation). The inputs are the ALUOp and funct fields. Only the entries for which the ALU control is asserted are shown. Some don't-care entries have been added. For example, the ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01. While we show all 10 bits of funct fields, note that the only bits with different values for the four R-format instructions are bits 30, 14, 13, and 12. Thus, we only need these four funct field bits as input for ALU control instead of all 10.

Truth table. From logic, a representation of a logical operation by listing all the values of the inputs and then in each case showing what the resulting outputs should be

Don't care term. An element of a logical function in which the output does not depend on the values of all the inputs. Don't-care terms may be specified in different ways

Recap: Formats of the Four Inst. Classes

- The opcode field is always in bits 6:0
  - The funct3 field and funct7 field serve as an extended opcode field
- The first register operand is always in bit positions 19:15 (rs1) for R-type instructions and branch instructions
  - Also specifies the base register for load and store instructions
- The second register operand is always in bit positions 24:20 (rs2) for R-type instructions and branch instructions
  - Also specifies the register operand that gets copied to memory for store instructions
- Another operand can also be a 12-bit offset for branch or load-store instructions
- The destination register is always in bit positions 11:7 (rd) for R-type instructions and load instructions

Name (Bit position)	Fields 31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

FIGURE 4.14 The four instruction classes (arithmetic, load, store, and conditional branch) use four different instruction formats.

(a) Instruction format for **R-type arithmetic instructions** (opcode = 51_{ten}), which have three register operands: rs1,rs2, and rd. Fields rs1 and rd are sources, and rd is the destination. The ALU function is in the funct3 and funct7 fields and is decoded by the ALU control design in the previous section. The R-type instructions that we implement are add, sub, and, and or.

(b) Instruction format for **I-type load instructions** (opcode = 3_{ten}). The register rs1 is the base register that is added to the 12-bit immediate field to form the memory address. Field rd is the destination register for the loaded value.

(c) Instruction format for **S-type store instructions** (opcode = 35_{ten}). The register rs1 is the base register that is added to the 12-bit immediate field to form the memory address. (The immediate field is split into a 7-bit piece and a 5-bit piece.) Field rs2 is the source register whose value should be stored into memory.

(d) Instruction format for **SB-type conditional branch instructions** (opcode = 99_{ten}). The registers rs1 and rs2 compared. The 12-bit immediate address field is sign-extended, shifted left 1 bit, and added to the PC to compute the branch target address. Figures 4.17 and 4.18 give the rationale for the unusual bit ordering for SB-type

ISA vs. Hardware Design
(HW/SW Interface)

- Compared with MIPS, RISC-V has instruction formats
  - ➢that look more complicated but actually simplify the hardware
  - ➢This can even improve the clock cycle time of some RISC-V implementations, especially the pipelined versions
  - ➢compilers, assemblers, and debuggers hide details of the instruction format from the programmer
- Example of the store instruction format for MIPS
  - ➢The store instruction format and its impact on the datapath design
- Check the second example on your own
  - ➢P. 273~274, Figs. 4.16~4.18

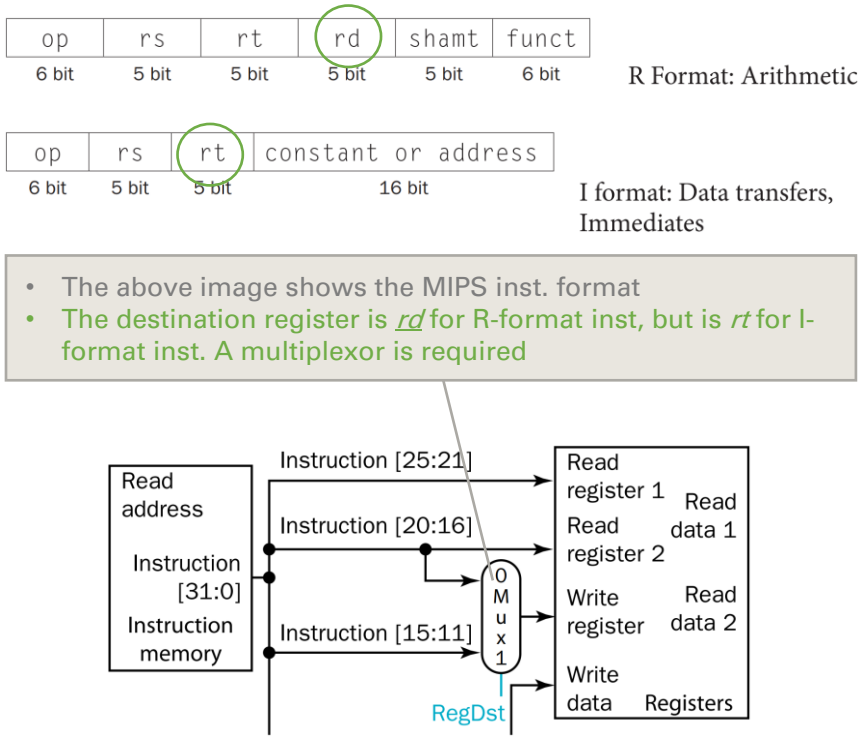


FIGURE 4.15 The MIPS, arithmetic instruction format, data transfer instruction format, and their impact on the MIPS datapath. For MIPS arithmetic instructions using the R format, rd is the destination register, rs is the first register operand, and rt is the second register operand. For MIPS load and immediate instructions, rs is still the first register operand, but rt is now the destination register. Hence the need of the 2:1 multiplexor to pick between the rd and rt fields to write the correct register

Setting of the Main Control Unit

- It will be useful to *try* to define the control function informally (Fig. 4.20) before we try to write a set of equations or a truth table for the control unit
- Fig. 4.22 defines how the control signals should be set for each opcode (7-bit opcode is its input)
  - this information follows directly from Figures 4.12, 4.20, and 4.21
  - Because the setting of the control lines depends only on the opcode, we define whether each control signal should be 0, 1, or don't care (X) for each of the opcode values (based on the given opcode)
- Please read the textbook for the descriptions of Figs. 4.19~4.22

Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

FIGURE 4.20 The effect of each of the six control signals. When the 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Remember that the state elements all have the clock as an implicit input and that the clock is used in controlling writes. Gating the clock externally to a state element can create timing problems. (See Appendix A for further discussion of this problem.)

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

FIGURE 4.22 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, and or). For all these instructions, the source register fields are rs1 and rs2, and the destination register field is rd; this defines how the signals ALUSrc is set. Furthermore, an R-type instruction writes a register (RegWrite = 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 fields. 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, RegWrite is set for a load to cause the result to be stored in the rd register. The ALUOp field for branch is set for 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. This type of don't care must be added by the designer, since it depends on knowledge of how the datapath works

Switch different inst. formats (e.g., R with 2 reg. sources and I with immediate)

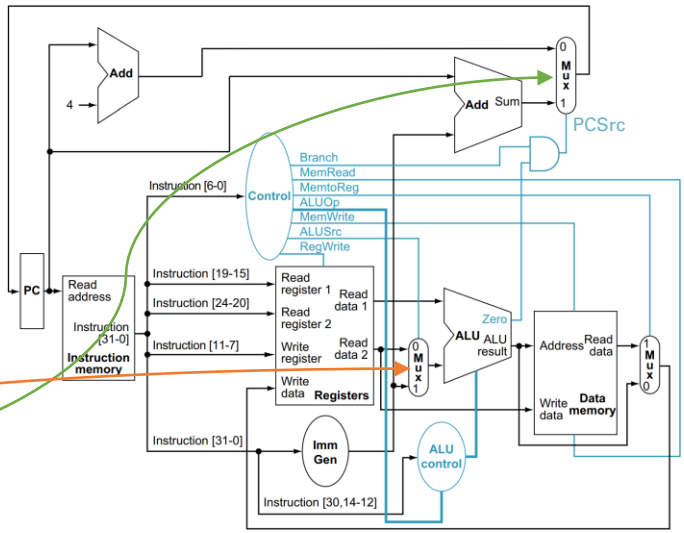


FIGURE 4.21 The simple datapath with the control unit. The input to the control unit is the 7-bit opcode field from the instruction. The outputs of the control unit consist of two 1-bit signals that are used to control multiplexors (ALUSrc and MemtoReg), three signals for controlling reads and writes 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. Notice that PCSrc is now a derived signal, rather than one coming directly from the control unit. Thus, we drop the signal name in subsequent figures

Execution Flow of R-Format Inst.

- The flow (four steps) of the datapath for handling an R-type instruction

- Use add x1, x2, x3 as an example

1. The instruction is fetched, and the PC is incremented
2. Two registers, x2 and x3, 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 portions of the opcode to generate the ALU function
4. The result from the ALU is written into the destination register (x1) in the register file

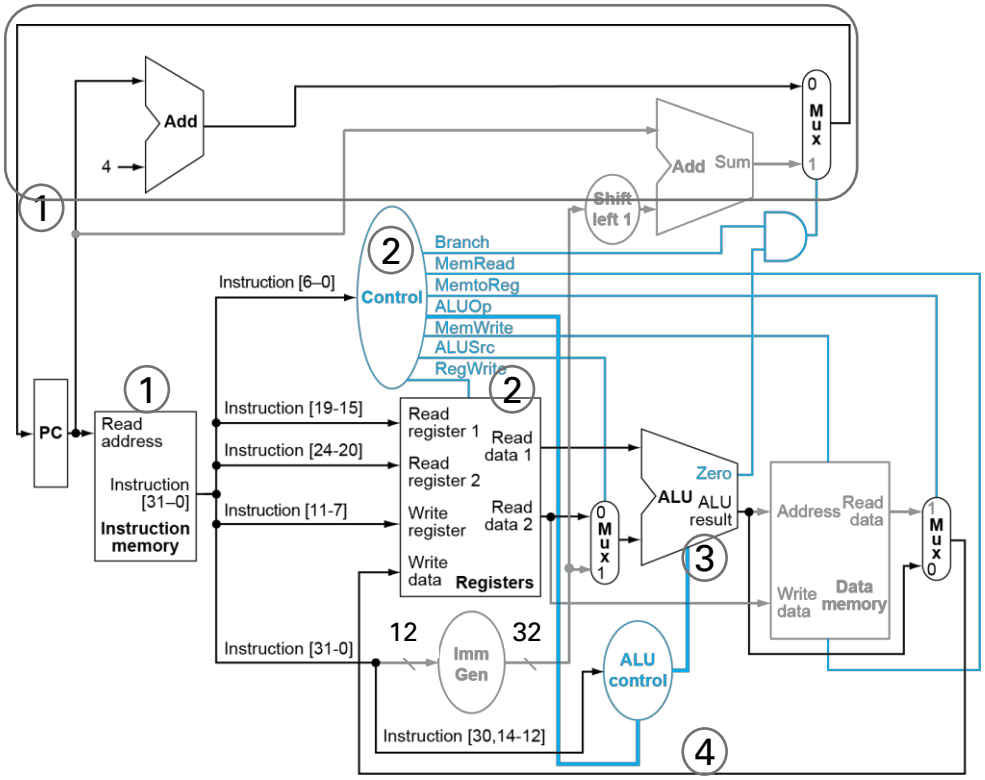


FIGURE 4.23 The datapath in operation for an R-type instruction, such as add x1, x2, x3. The control lines, datapath units, and connections that are active are highlighted

Execution Flow of I-Format Inst.

- The flow (five steps) of the datapath for handling an I-type instruction

- Use lw x1, offset(x2) as an example

1. An instruction is fetched from the instruction memory, and the PC is incremented
2. A register (x2) value is read from the register file; compute control signals
3. The ALU computes the sum of the value read from the register file and the sign-extended 12 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 (x1)

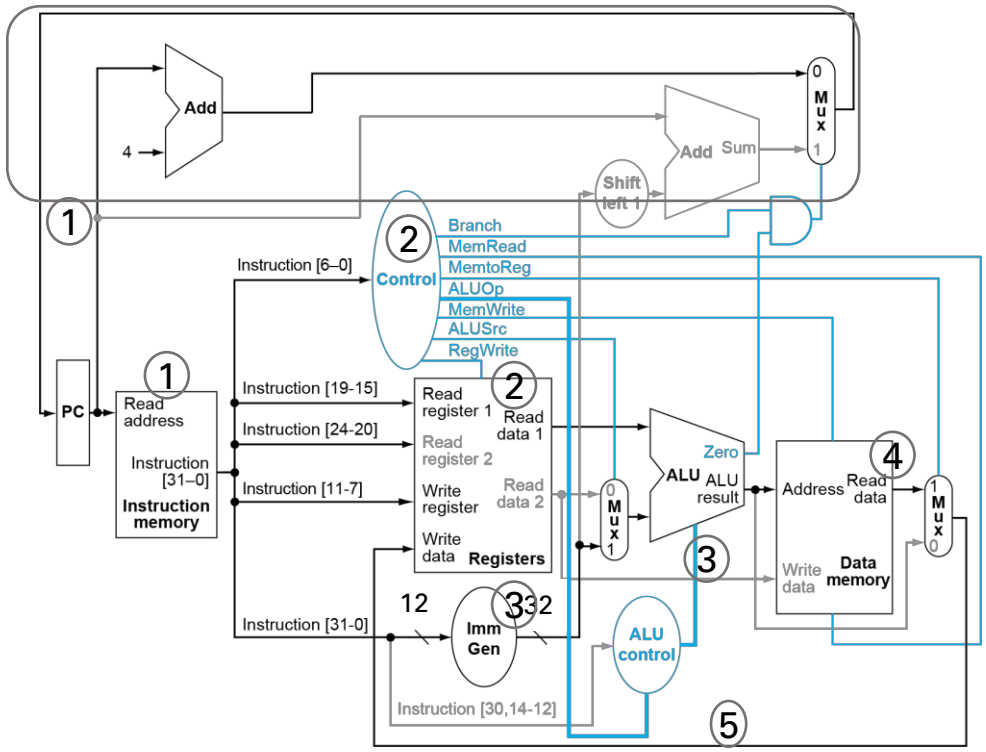


FIGURE 4.24 The datapath in operation for a load instruction. The control lines, datapath units, and connections that are active are highlighted. A store instruction would operate very similarly. The main difference would be that the memory control would indicate a write rather than a read, the second 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.

Execution Flow of Branch Inst.

- The flow (four steps) of the datapath for handling a branch-if-equal instruction

- Use beq x1, x2, offset as an example

1. An instruction is fetched from the instruction memory, and the PC is incremented
2. Two registers, x1 and x2, are read from the register file
3. The ALU subtracts one data value from the other data value, both read from the register file
   - ➢ The value of PC is added to the sign-extended, 12 bits of the instruction (offset) left shifted by one; the result is the branch target address
4. The Zero status information from the ALU is used to decide which adder result to store in the PC

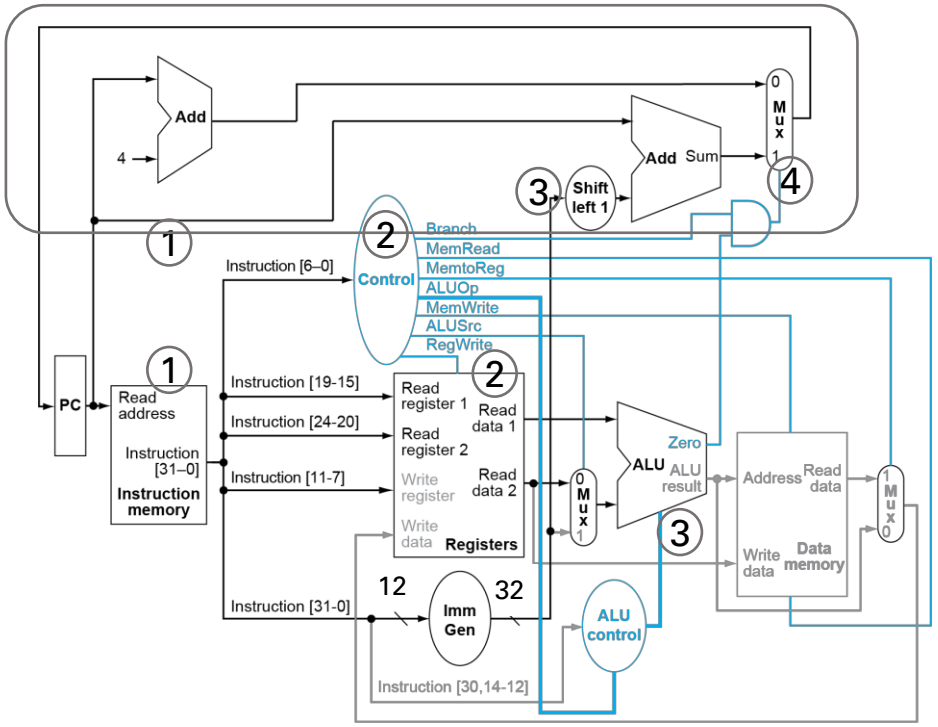


FIGURE 4.25 The datapath in operation for a branch-if-equal instruction. The control lines, datapath units, and connections that are active are highlighted. After using the register file and ALU to perform the compare, the Zero output is used to select the next program counter from between the two candidates

The Logic of the Control Unit

- The logic is represented as the large truth table in Fig. 4.26
  - ➢ The inputs of the control unit are the opcode bits of an inst. (I[0-6])
  - ➢ The outputs are the control lines
    - ❖ The contents are listed in Fig. 4.22

Performance issue of the single cycle design

- Longest delay determines the clock period
  - ➢ Critical path is the load instruction, which takes five steps to complete
    - ❖ Instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
- Violates the design principle
  - ➢ Making the common case fast
  - ➢ The single cycle design is bounded by the clock cycle of a load instruction
- The performance is improved by the pipelining mechanism

Inputs (I[0-6])		1100110	1100000	1100010	1100011
Input or output	**Signal name**	**R-format**	**lw**	**sw**	**beq**
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	0
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

FIGURE 4.26 The control function for the simple single-cycle implementation is completely specified by this truth table. The top seven rows of the table gives the combinations of input signals that correspond to the four instruction classes, one per column, that determine the control output settings. The bottom portion of the table gives the outputs for each of the four opcodes. Thus, the output RegWrite is asserted for two different combinations of the inputs. If we consider only the four opcodes shown in this table, then we can simplify the truth table by using don't cares in the input portion. For example, we can detect an R-format instruction with the expression Op4 · Op5, since this is sufficient to distinguish the R-format instructions from lw, sw, and beq. We do not take advantage of this simplification, since the rest of the RISC-V opcodes are used in a full implementation

Pipelining

- An implementation technique
  - in which multiple instructions are overlapped in execution, much like an assembly line
- Explore the anatomy of a pipelined computer
  - Details in sections 4.7 through 4.10

- The laundry analogy for *pipelining*
- Imaging that a laundry involves the following steps
  1. Place one dirty load of clothes in the washer
  2. When the washer is finished, place the wet load in the dryer
  3. When the dryer is finished, place the dry load on a table and fold
  4. When folding is finished, ask your roommate to put the clothes away (in a closet)

Pipelining (Cont'd)

- Sequential laundry
  - ➢One step (stage) at a time
  - ➢Each step takes a half hour
  - ➢Time for four loads = 8 hrs
    - ❖2hr (one load) * 4 (four loads: Ann, Brain, Cathy, and Don)

- Pipelined laundry
  - ➢Time for four loads = 3.5 hrs
  - ➢Speedup = 2.3 (8/3.5)
  - ➢Ideal speedup (with sufficient load) = 4 (# of stages)

- Pipeline improves throughput via overlapping execution
  - ➢But, it does not decrease the time to complete one load

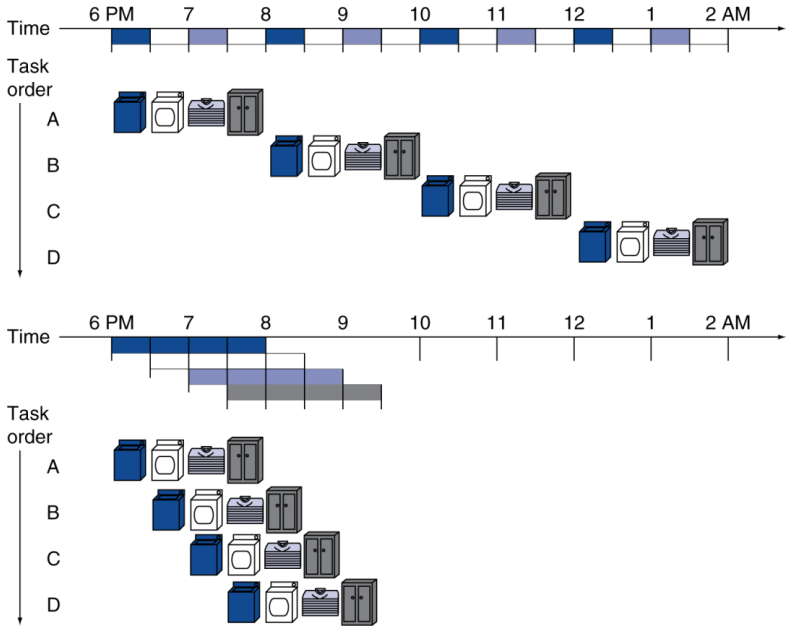


FIGURE 4.27 The laundry analogy for pipelining. Ann, Brian, Cathy, and Don each have dirty clothes to be washed, dried, folded, and put away. The washer, dryer, "folder," and "storer" each take 30 minutes for their task. Sequential laundry takes 8 hours for four loads of washing, while pipelined laundry takes just 3.5 hours. We show the pipeline stage of different loads over time by showing copies of the four resources on this two-dimensional time line, but we really have just one of each resource

- As soon as the washer is finished with the first load and placed in the dryer, you load the washer with the second dirty load
- When the first load is dry, you place it on the table to start folding, move the wet load to the dryer, and put the next dirty load into the washer
- Next, you have your roommate put the first load away, you start folding the second load, the dryer has the third load, and you put the fourth load into the washer

RISC-V Pipeline

- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand (for load/store inst.)
  5. WB: Write result back to register

- In the rest of this chapter, we limit our attention to seven instructions:
  1. load word (lw),
  2. store word (sw),
  3. add (add),
  4. subtract (sub),
  5. AND (and),
  6. OR (or), and
  7. branch if equal (beq)

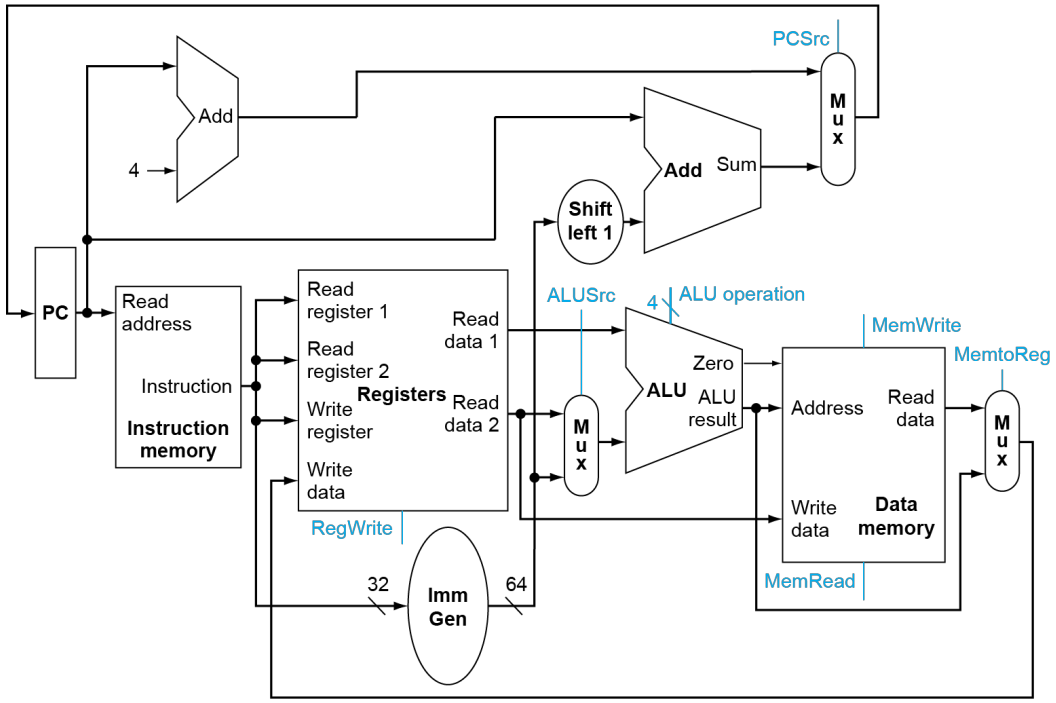


FIGURE 4.11 The simple datapath for the core RISC-V architecture combines the elements required by different instruction classes. The components come from Figures 4.6, 4.9, and 4.10. This datapath can execute the basic instructions (load-store register, ALU operations, and branches) in a single clock cycle. Just one additional multiplexor is needed to integrate branches

Example: Single Cycle vs. Pipeline Performance I

- The pipelined model
  - Assume that the operation times for the major functional units in this example:
  - 200 ps for memory access for instructions or data,
  - 200 ps for ALU operation, and
  - 100 ps for register file read or write
- The single-cycle model
  - every instruction takes exactly one clock cycle, so the clock cycle must be stretched to accommodate the slowest instruction
- Table below shows the time for each of the seven inst.

- The single-cycle design must allow for the slowest instruction (i.e., lw)
- The time required for every instruction is 800ps

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

FIGURE 4.28 Total time for each instruction calculated from the time for each component. This calculation assumes that the multiplexors, control unit, PC accesses, and sign extension unit have no delay

Example: Single Cycle vs. Pipeline Performance II

- Compare nonpipelined and pipelined execution of three load register instructions
- Non-pipelined
  - The time between the first and fourth instructions in the nonpipelined design is 2,400ps (3 × 800ps)
  - The slowest inst. takes 800ps
- Pipelined
  - Each pipeline stage takes a single cycle
  - The clock cycle must be long enough to accommodate the slowest operation (stage)
  - The worst-case clock cycle is 200ps
  - The time between the first and fourth instruction is 600ps (3 × 200ps)

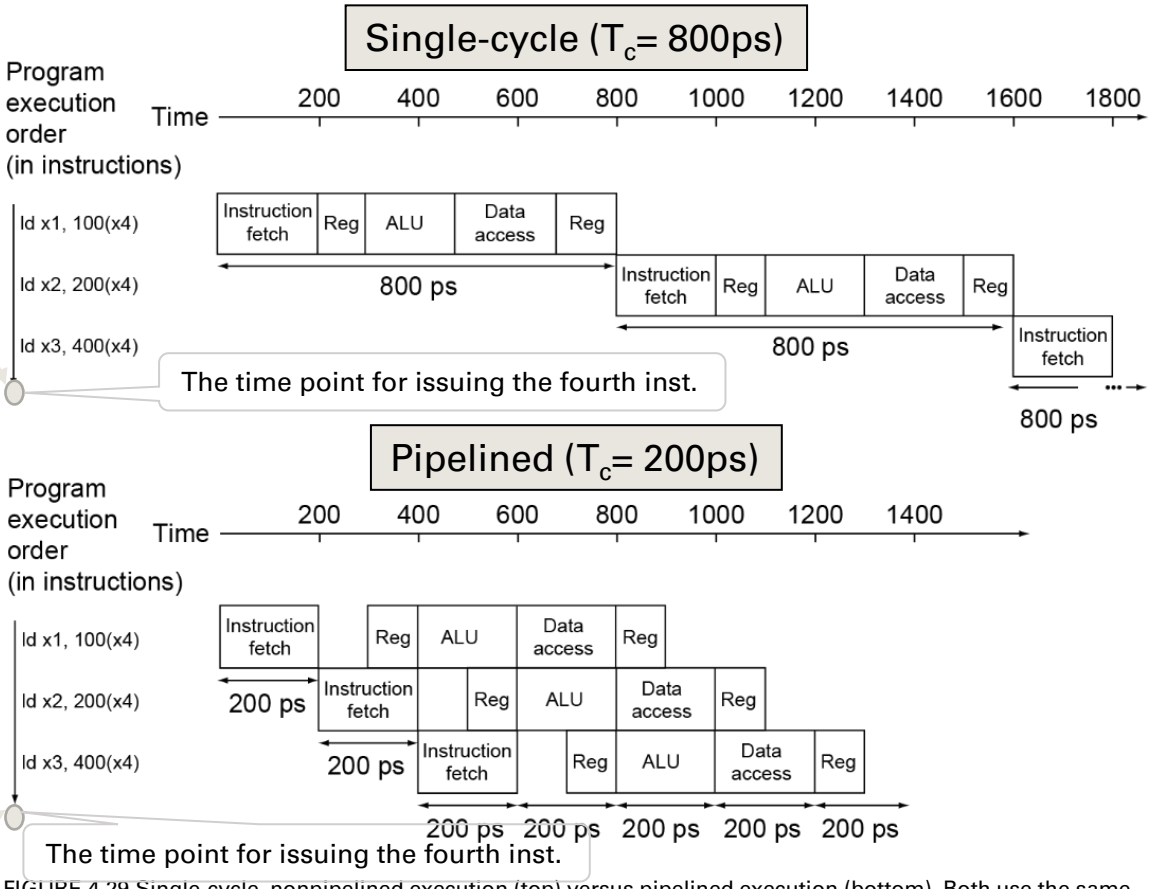


FIGURE 4.29 Single-cycle, nonpipelined execution (top) versus pipelined execution (bottom). Both use the same hardware components, whose time is listed in Figure 4.28. In this case, we see a fourfold speed-up on average time between instructions, from 800ps down to 200ps. Compare this figure to Figure 4.27. For the laundry, we assumed all stages were equal. If the dryer were slowest, then the dryer stage would set the stage time. The pipeline stage times of a computer are also limited by the slowest resource, either the ALU operation or the memory access. We assume the write to the register file occurs in the first half of the clock cycle and the read from the register file occurs in the second half. We use this assumption throughout this chapter.

Example: Single Cycle vs. Pipeline Performance III

- If all stages are balanced
  - I.e., all stages take the same time
  - Time between instructions$_{\text{pipelined}}$ = $\dfrac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$
  - E.g., ~5 times faster for a five-stage pipeline processor
- If not balanced, speedup is less

- Speedup due to increased throughput in a pipelined processor
  - Latency (time for each instruction) does not decrease
  - Latency of an instruction is the same
  - One instruction is completed every 200ps

- Nonpipelined time between insts. = 1,000ps (at the bottom of Fig. 4.29)
- Five stages
- Pipelined time between insts. = 200ps

Insights of the Design for Pipelined Execution

RISC-V ISA designed for pipelining

- Fixed length of RISC-V instructions
  - All instructions are 32-bits
  - Easier to fetch and decode in one cycle
  - cf. x86 with 1- to 15-byte variable length instructions; in practice x86 insts are translated into simpler micro-code (like RISC-V insts) before the execution in the pipeline
- Few and regular instruction formats
  - with the source and destination register fields being located in the same place in each instruction
  - Can decode and read registers in one step
- Addressing of Load/Store insts.
  - Memory operands only appear in loads or stores in RISC-V
  - This design allows to calculate address in the 3rd stage (execution) and then, access memory in the 4th stage

Pipeline Hazards

- Hazards
  - There are situations (events) that prevent starting the next instruction in the next cycle

Three different types of hazards

- Structure hazards
  - The hardware cannot support the combination of instructions that are set to execute in the same clock cycle
  - A required resource is busy
- Data hazard
  - The pipeline must be stalled because one step must wait for another to complete
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Arising from the need to make a decision based on the results of one instruction while others are executing
    - The flow of instruction addresses is not what the pipeline expected
  - Deciding on a control action depends on previous instruction

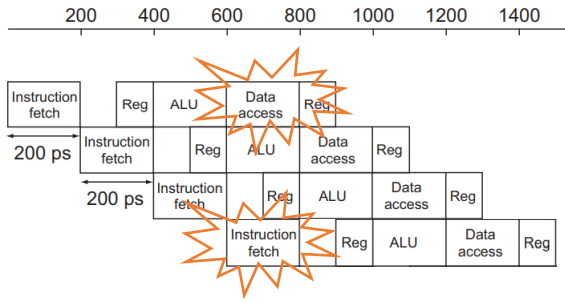
Structure Hazard

- Conflict for use of a resource
  - ➢Solution: allocate more resources

- Suppose the RISC-V pipeline with a single memory (instead of two memories)
  - ➢The 1st inst. requires data access
  - ➢The fetch of the 4th inst. from the same memory would have to stall for that cycle
  - ➢Would cause *a* pipeline "bubble"
    - ❖How many bubbles are needed in the right image?

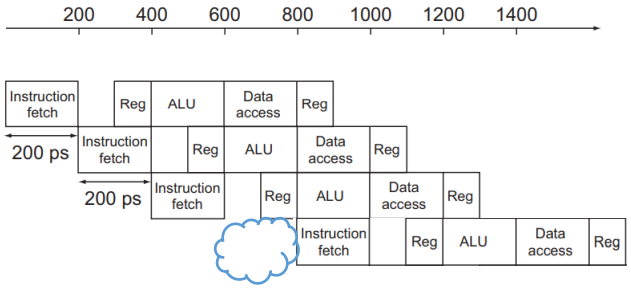
- Pipelined datapaths require separate instruction/data memories
  - ➢Or separate instruction/data caches

A fourth instruction causes a memory access conflict



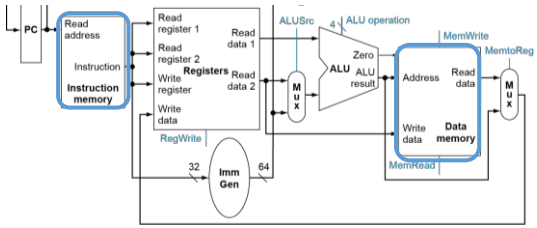
Add a fourth inst. into FIGURE 4.29

A pipeline bubble example



Add a bubble for the fourth inst.

Separate mem. solution



Graphical Representation of the Pipeline

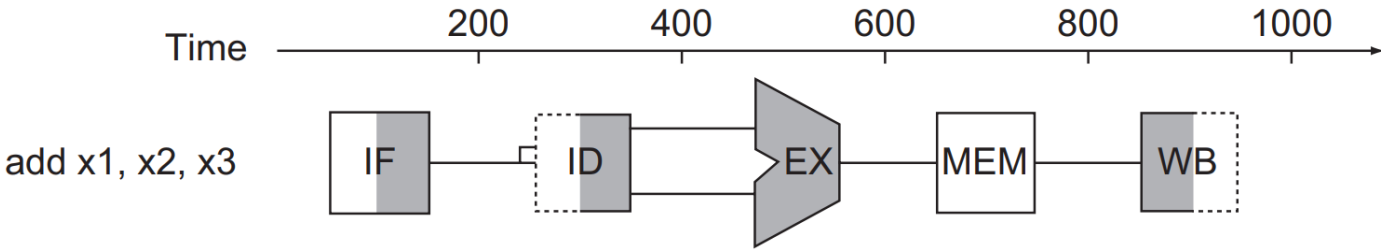


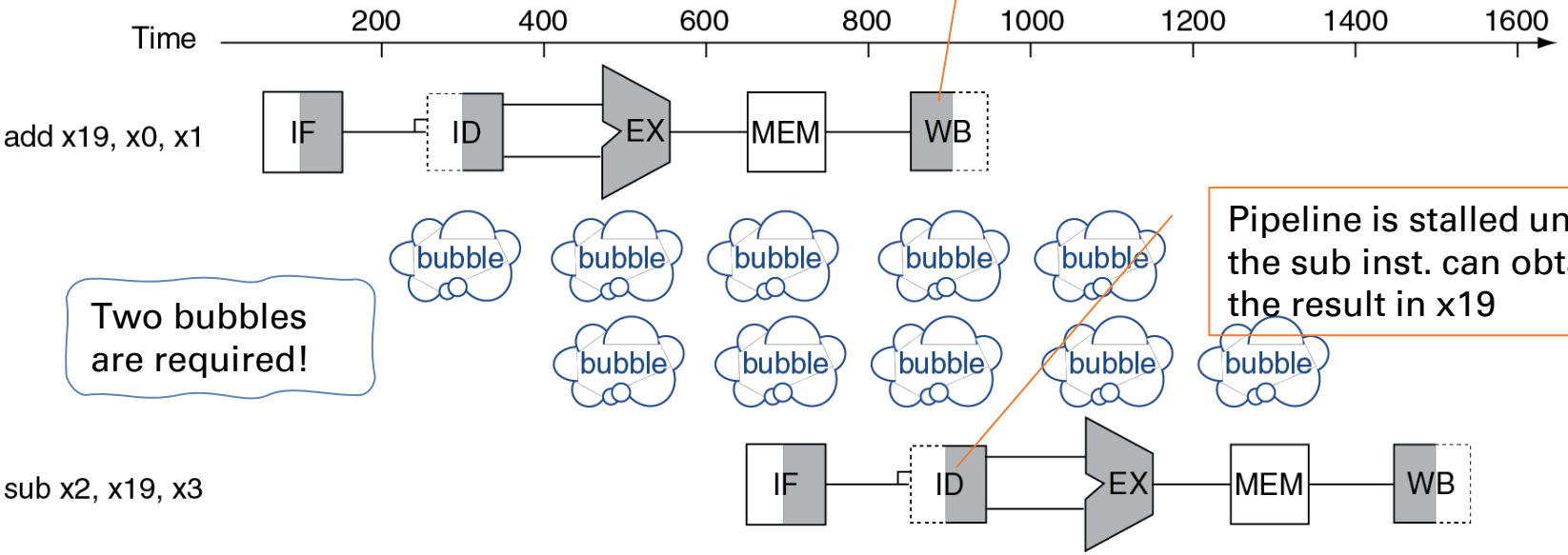
FIGURE 4.30 Graphical representation of the instruction pipeline, similar in spirit to the laundry pipeline in Figure 4.27. Here we use symbols representing the physical resources with the abbreviations for pipeline stages used throughout the chapter. The symbols for the five stages: IF for the instruction fetch stage, with the box representing instruction memory; ID for the instruction decode/ register file read stage, with the drawing showing the register file being read; EX for the execution stage, with the drawing representing the ALU; MEM for the memory access stage, with the box representing data memory; and WB for the write-back stage, with the drawing showing the register file being written.
The shading indicates the element is used by the instruction. Hence, MEM has a white background because add does not access the data memory.
Shading on the right half of the register file or memory means the element is read in that stage, and shading of the left half means it is written in that stage. Hence the right half of ID is shaded in the second stage because the register file is read, and the left half of WB is shaded in the fifth stage because the register file is written.

Data Hazards (Bubbles)

- An instruction depends on completion of data access by an earlier instruction that is still in the pipeline

```
add     x19, x0, x1
sub     x2, x19, x3
```

The add instruction doesn't write its result until the fifth stage



Data Hazards (Sol 1: Forwarding)

- Re-ordering the instructions is one possible solution
  - which relies on *compilers* to remove all such hazards
  - But, the results would not be satisfactory
- These dependences happen just too often and the delay is far too long to expect the compiler to rescue us from this dilemma
- Observation: Use result when it is computed
  - Don't wait for it to be stored in a register
  - E.g., for the example in the previous page, we can supply the data of x19 as an input to sub, as soon as the ALU creates the sum for the add
- Solution: Forwarding (Bypassing)
  - Adding extra hardware (in Fig. 4.31) to retrieve the missing item early from the internal resources

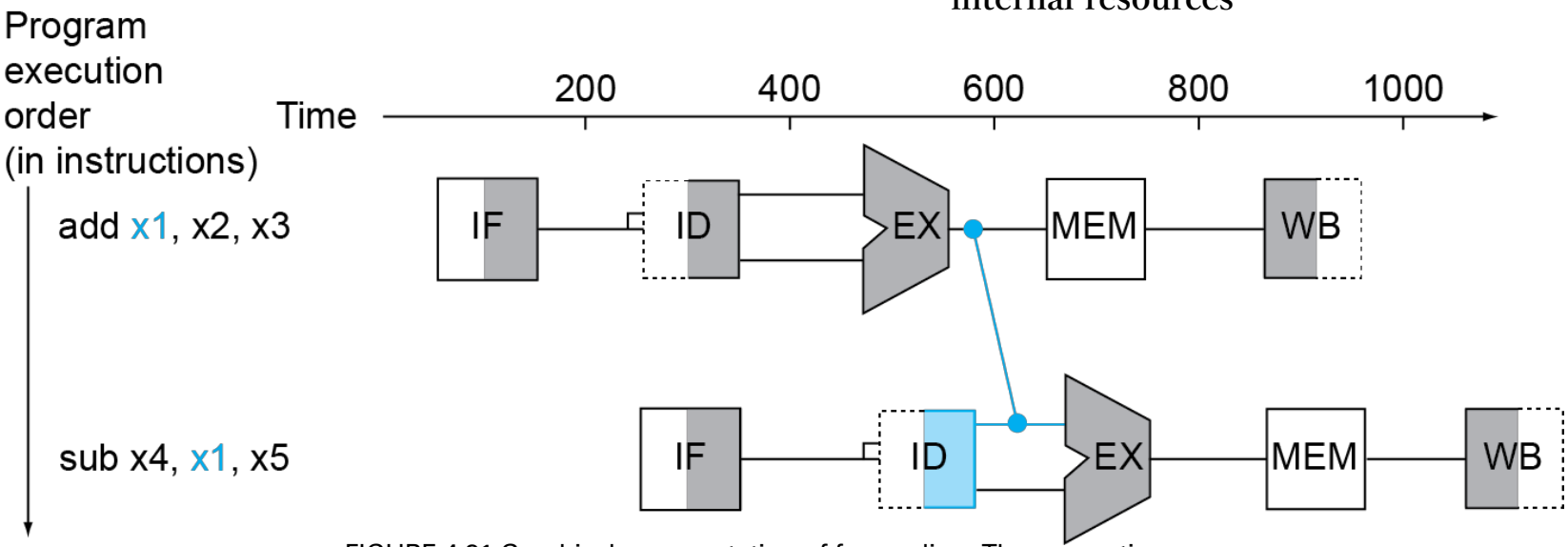


FIGURE 4.31 Graphical representation of forwarding. The connection shows the forwarding path from the output of the EX stage of add to the input of the EX stage for sub, replacing the value from register x1 read in the second stage of sub.

Load-use Data Hazard

- Forwarding cannot always avoid stalls
- Using the load and sub insts sequence as an example
  - ➢The desired data would be available only
  - ➢after the fourth stage of the first ld instruction in the dependence,
  - ➢which is too late for the input of the third stage of sub
  - ➢Can't forward backward in time!
  - ➢Need to stall (bubble) one cycle
  - → Load-use data hazard
- Sec. 4.8 details the issue
  - ➢with SW and HW solutions

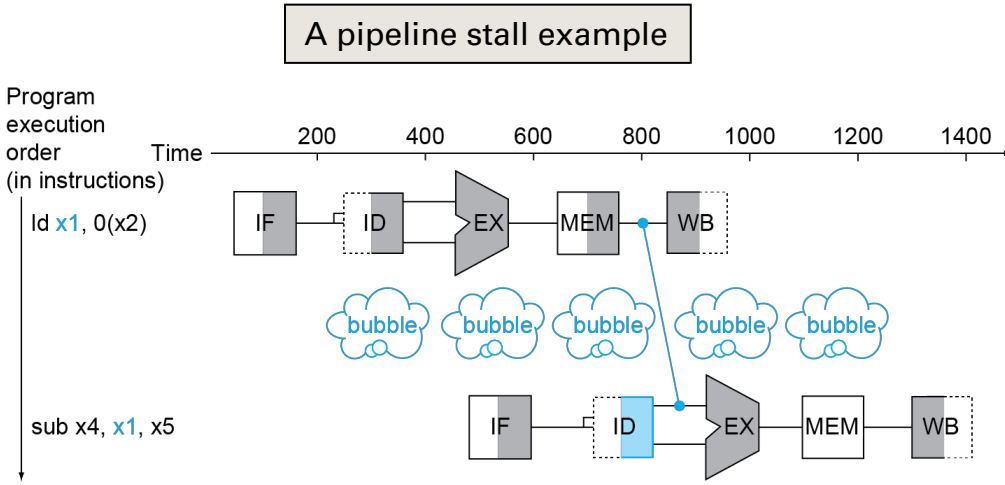


FIGURE 4.32 We need a stall even with forwarding when an R-format instruction following a load tries to use the data. Without the stall, the path from memory access stage output to execution stage input would be going backward in time, which is impossible. This figure is actually a simplification, since we cannot know until after the subtract instruction is fetched and decoded whether or not a stall will be necessary. Section 4.7 shows the details of what really happens in the case of a hazard

Data Hazards (Sol 2: Code Scheduling)

- Reorder code to avoid use of load result in the next instruction
- Assembly code for C statements

```
a = b + e;
c = b + f;
```

Both add instructions have a hazard because of their respective dependence on the previous lw instruction

- Notice that forwarding eliminates several other potential hazards, including the dependence of the first add on the first lw and any hazards for store instructions
- Moving up the third lw instruction to become the third instruction eliminates both hazards

```
                ld   x1, 0(x31)
                ld   x2, 8(x31)
stall  →        add  x3, x1, x2
                sd   x3, 24(x31)
                ld   x4, 16(x31)
stall  →        add  x5, x1, x4
                sd   x5, 32(x31)
```

13 cycles

move up

```
ld   x1, 0(x31)
ld   x2, 8(x31)
ld   x4, 16(x31)
add  x3, x1, x2
sd   x3, 24(x31)
add  x5, x1, x4
sd   x5, 32(x31)
```

11 cycles

Control Hazard (Branch Hazard)

- The fetched instruction is not the one that is needed
  - when the proper instruction cannot execute in the proper pipeline clock cycle
  - Solutions are proposed to improve the efficiency

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
- Pipeline can't always fetch correct instruction
  - Still working on ID stage of branch

- In RISC-V pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

Control Hazard (Sol 1: Stall on Branch)

- One intuitive solution is to *stall* immediately after we fetch a branch
  - ➢ waiting until the pipeline determines the outcome of the branch and
  - ➢ knows what instruction address to fetch from

- Notice that we must begin fetching the instruction following the branch on the following clock cycle
  - ➢ Nevertheless, the pipeline cannot possibly know what the next instruction should be, since it only just received the branch instruction from memory
  - ➢ Without proper HW assisting, the next instruction is determined at the third stage EX of the pipeline when handling the control branch

- IF we add extra hardware so that we can test a register, calculate the branch address, and update the PC during the second stage of the pipeline (see Sec. 4.9 for details)

- As a result, the pipeline involving conditional branches looks like Figure 4.33
  - ➢ The branch is taken and the branch target (or inst.) is fetched (400ps later than the beq inst.)

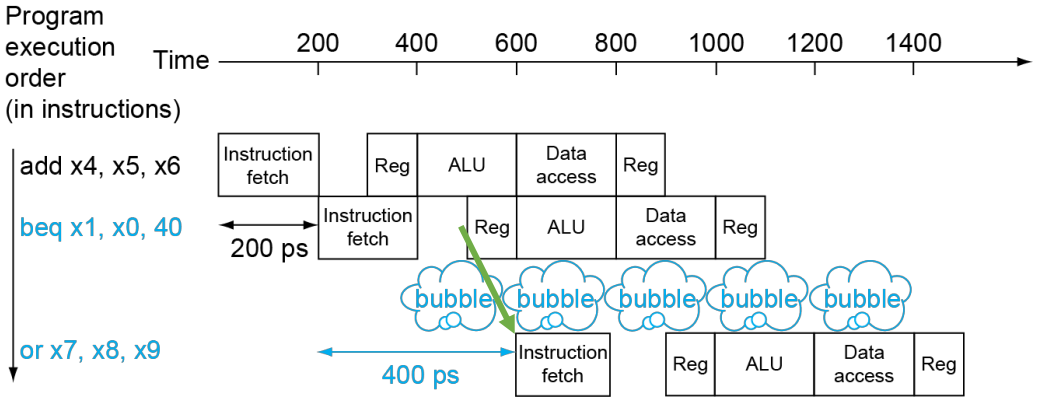


FIGURE 4.33 Pipeline showing stalling on every conditional branch as solution to control hazards. This example assumes the conditional branch is taken, and the instruction at the destination of the branch is the "or" instruction. There is a one-stage pipeline stall, or bubble, after the branch. In reality, the process of creating a stall is slightly more complicated, as we will see in Section 4.9. The effect on performance, however, is the same as would occur if a bubble were inserted

Control Hazard (Sol 2: Branch Prediction)

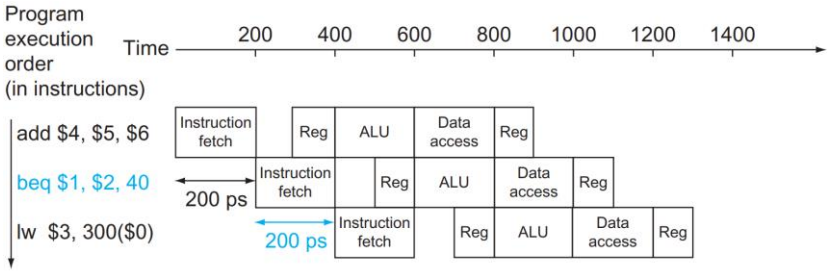
- To prevent the pipeline stalls, branch prediction is adopted as a second solution (as in modern processors)

- Prediction is one of the great ideas from Ch. 1
  - If you're pretty sure you have the right formula to wash uniforms, then
  - just predict that it will work and wash the second load while waiting for the first load to dry
- If a branch cannot be resolved in the second stage, an even larger slowdown occurs
  - The overhead grows with the depth of the pipeline

- Prediction is correct, no slow down
- Prediction is wrong, need to redo the load that was washed while guessing the decision

- Figure 4.32 assumes "untaken" branch predictor is used
- That is, the inst. (lw) following the beq is fetched by default
- If the branch is taken (mis-prediction), the pipeline stalls for one cycle before the branch target inst. (or) is fetched

The pipeline for the untaken branch



The pipeline for a taken branch (mis-predict; one bubble)

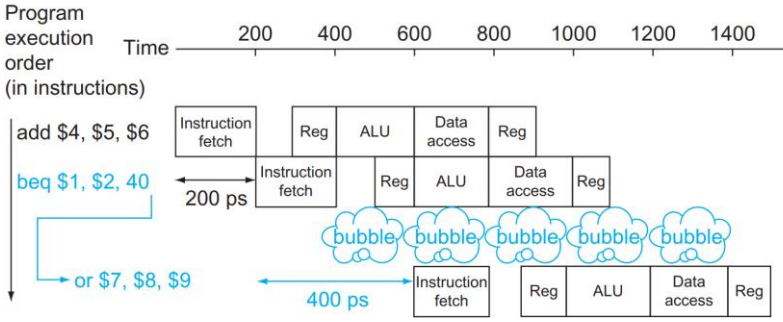


FIGURE 4.32 Predicting that branches are not taken as a solution to control hazard. The top drawing shows the pipeline when the branch is not taken. The bottom drawing shows the pipeline when the branch is taken. As we noted in Figure 4.31, the insertion of a bubble in this fashion simplifies what actually happens, at least during the first clock cycle immediately following the branch. Section 4.8 will reveal the details

Static vs. Dynamic Branch Prediction

- Static branch prediction
  - ➢Based on typical branch behavior
  - ➢Example: loop and if-statement branches
  - ➢Predict *backward* branches taken
  - ➢Predict *forward* branches not taken
  - ➢The inner branch mis-predicts the first and last iterations
    - ❖Leads to an 80% hit ratio

An example loop

```
for (i=0; i<10; i++)
  for (j=0; j<10; j++)
    ...
```

- Dynamic branch prediction
  - ➢Hardware measures actual (runtime) branch behavior
  - ➢E.g., keeping a history of recent branches
  - ➢Assume future behavior will continue the trend (kept in the history)
    - ❖When wrong, stall while re-fetching, and update history

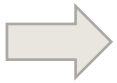
Control Hazard (Sol 3: Delayed Branch)

- The delayed branch always executes the next sequential instruction,
  - ➢with the branch taking place after that one instruction delay

- MIPS SW (e.g., compiler) will place an instruction immediately
  - ➢after the delayed branch instruction that is not affected by the branch, and
  - ➢a taken branch changes the address of the instruction that follows this safe instruction
  - ➢It is hidden from the programmer because the compiler/assembler can automatically arrange the instructions to get the branch behavior desired by the programmer

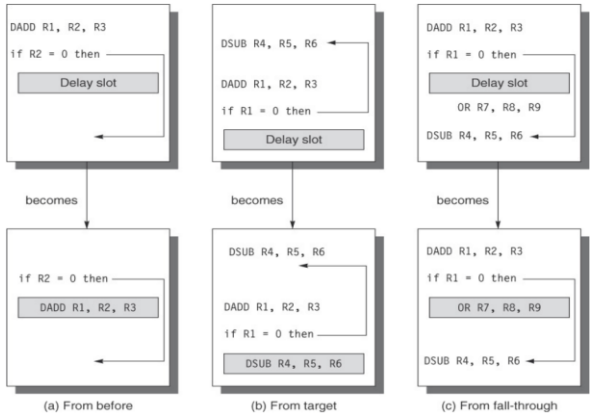
- Use the example in Fig. 4.33
  - The add instruction before the branch does not affect the branch and
  - can be moved after the branch (called delay slot) to hide the branch delay fully
- Since delayed branches are useful when the branches are short, it is rare to see a processor with a delayed branch of more than one cycle
- For longer branch delays, hardware-based branch prediction is usually used

```
add $4, $5, $6
beq $1, $2, 40
lw $3, 300($0)
```



```
beq $1, $2, 40
add $4, $5, $6
lw $3, 300($0)
```

Choose the inst. for delay slot under three different branch directions: From before, From target, and From fall-through



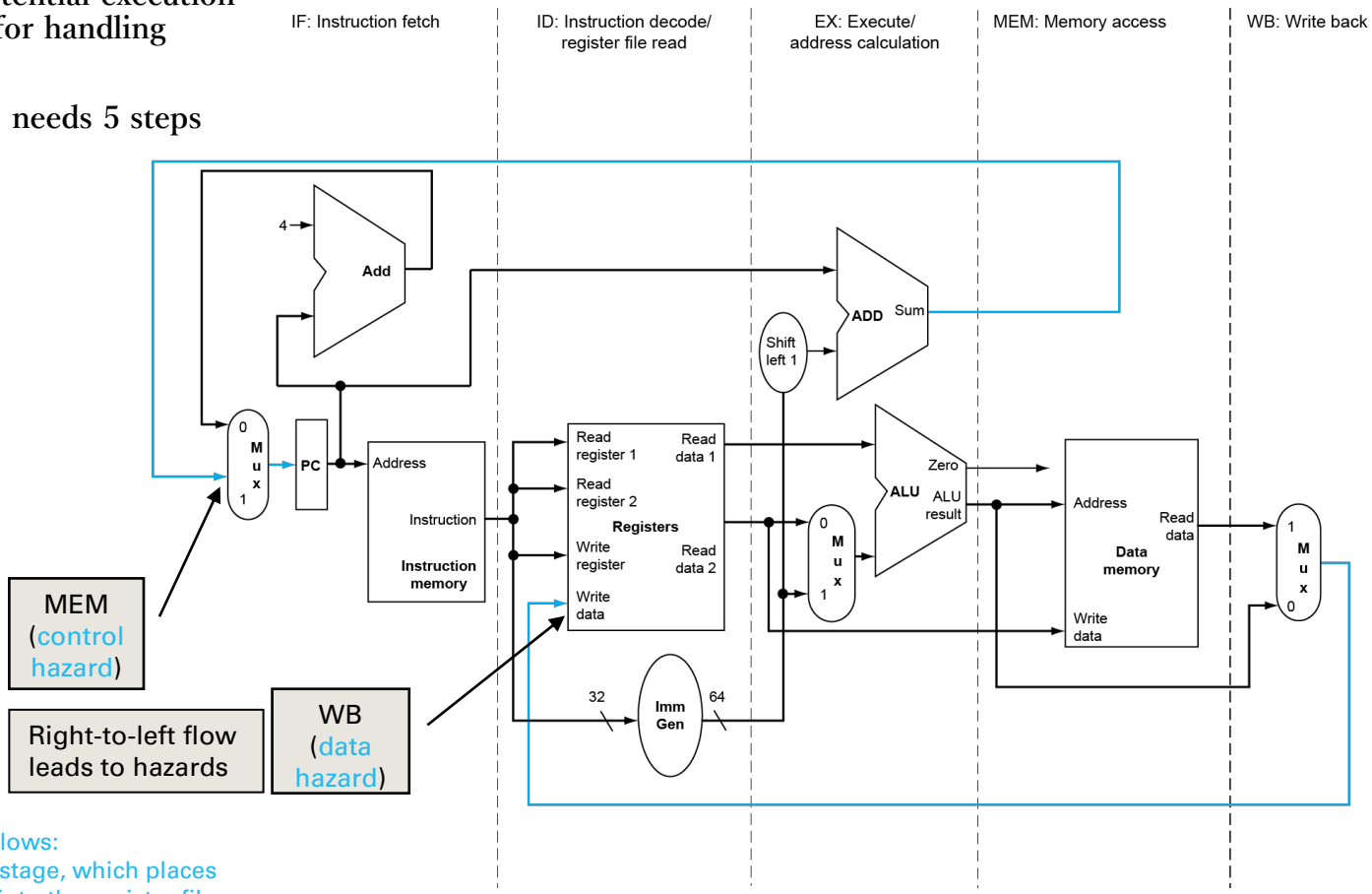
A Quick View of the Following Contents

- Sec. 4.7
  - examines the design of a pipelined datapath and the basic control
  - understands how pipelining is implemented and the challenges of dealing with hazards
- Sec. 4.8
  - explores the implementation of forwarding and stalls
- Sec. 4.9
  - learns more about solutions to branch hazards
- Sec. 4.10
  - finally sees how exceptions are handled

Stages of the Single-cycle Datapath

- The single-cycle datapath is broken into potential execution stages (steps) for handling instructions
- Not every inst. needs 5 steps

The single-cycle datapath of five stages of instruction execution (Sec. 4.4)



Two right-to-left flows:

1. The write-back stage, which places the result back into the register file in the middle of the datapath
2. The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage

FIGURE 4.35 The single-cycle datapath from Section 4.4 (similar to Figure 4.21). Each step of the instruction can be mapped onto the datapath from left to right. The only exceptions are the update of the PC and the write-back step, shown in color, which sends either the ALU result or the data from memory to the left to be written into the register file. (Normally we use color lines for control, but these are data lines.)

The Pipelined Datapath

- All instructions must update some state in the processor
  - i.e., the register file, memory, or the PC, so a separate pipeline register is redundant to the state that is updated
  - For example, a load instruction will place its result in one of the 32 registers, and
  - any later instruction that needs that data will simply read the appropriate register
- Notice that there is no pipeline register at the end of the write-back stage

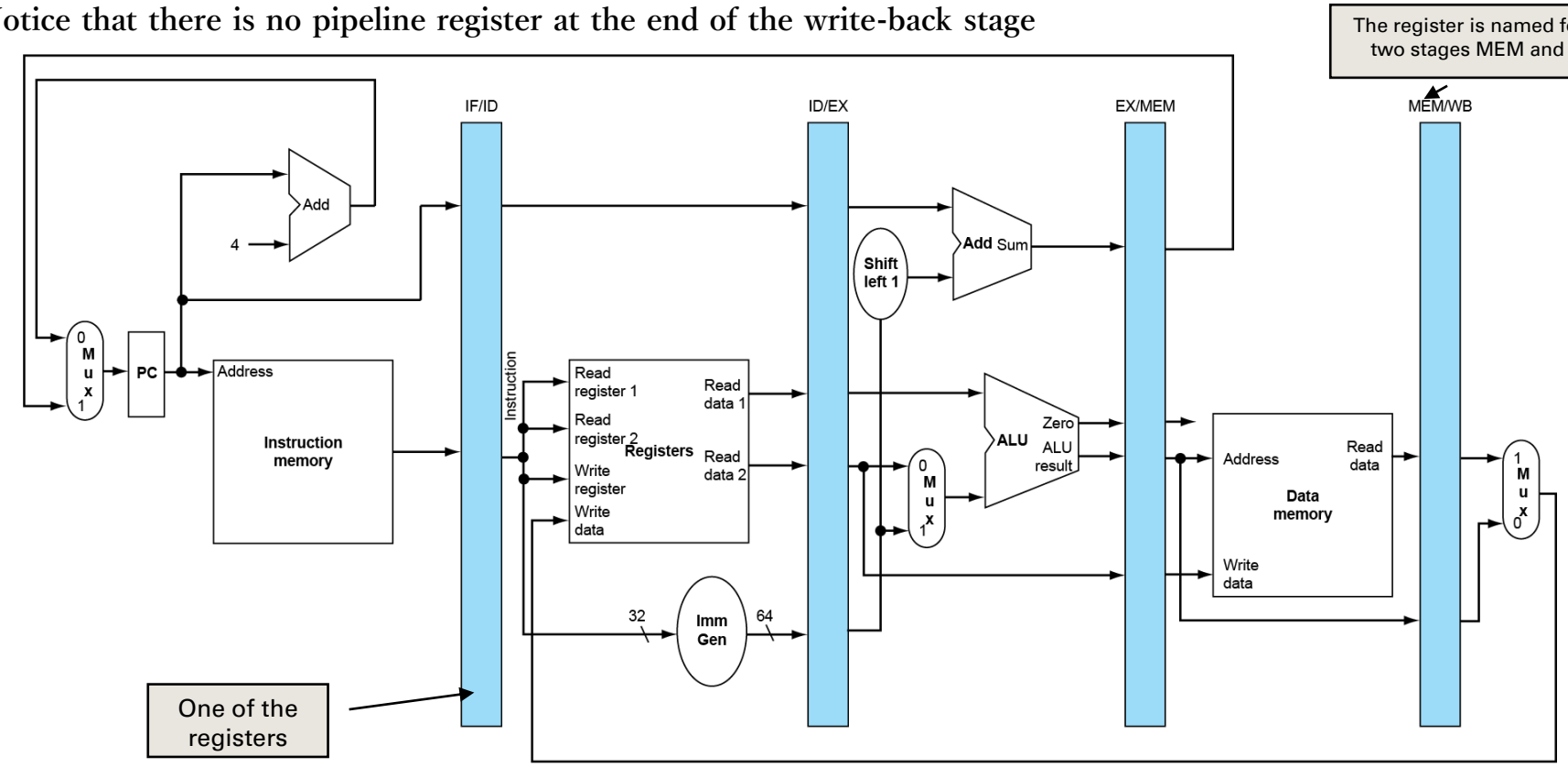


FIGURE 4.37 The pipelined version of the datapath in Figure 4.35. The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled IF/ID because it separates the instruction fetch and instruction decode stages. The registers must be wide enough to store all the data corresponding to the lines that go through them. For example, the IF/ID register must be 96 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 64-bit PC address. We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 256, 193, and 128 bits, respectively

Highlight IF and ID Stages of Load Inst.

IF

- The top image shows the instruction being read from memory using the address in the PC and then being placed in the IF/ID pipeline register
- The PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle
- This PC is also saved in the IF/ID pipeline register in case it is needed later for an instruction, such as beq
- The computer cannot know which type of instruction is being fetched, so it must prepare for any instruction, passing potentially needed information down the pipeline

Instruction decode (ID) & register file read

- The bottom image shows the instruction portion of the IF/ID pipeline register supplying the immediate field, which is sign-extended to 64 bits, and the register numbers to read the two registers
- All three values are stored in the ID/EX pipeline register, along with the PC address
- We again transfer everything that might be needed by any instruction during a later clock cycle

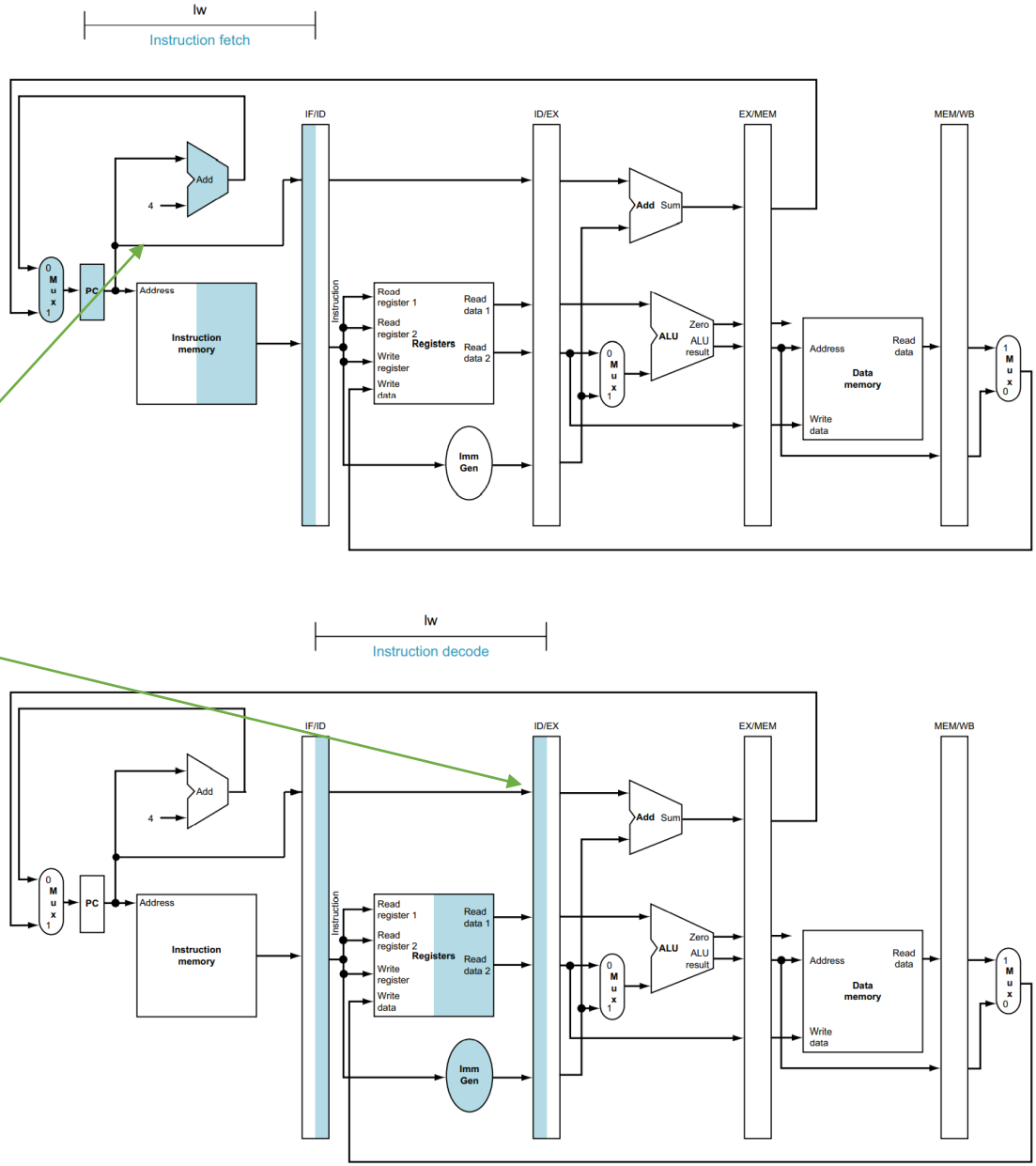


FIGURE 4.38 IF and ID: First and second pipe stages of an instruction, with the active portions of the datapath in Figure 4.37 highlighted. The highlighting convention is the same as that used in Figure 4.30. As in Section 4.2, there is no confusion when reading and writing registers, because the contents change only on the clock edge. Although the load needs only the top register in stage 2, it doesn't hurt to do potentially extra work, so it sign-extends the constant and reads both registers into the ID/EX pipeline register. We don't need all three operands, but it simplifies control to keep all three

Highlight EX Stage of Load Inst.

Execute (EX) or address calculation

- The image shows that the load instruction reads the contents of a register and the sign-extended immediate from the ID/EX pipeline register and adds them using the ALU
- That sum is placed in the EX/MEM pipeline register

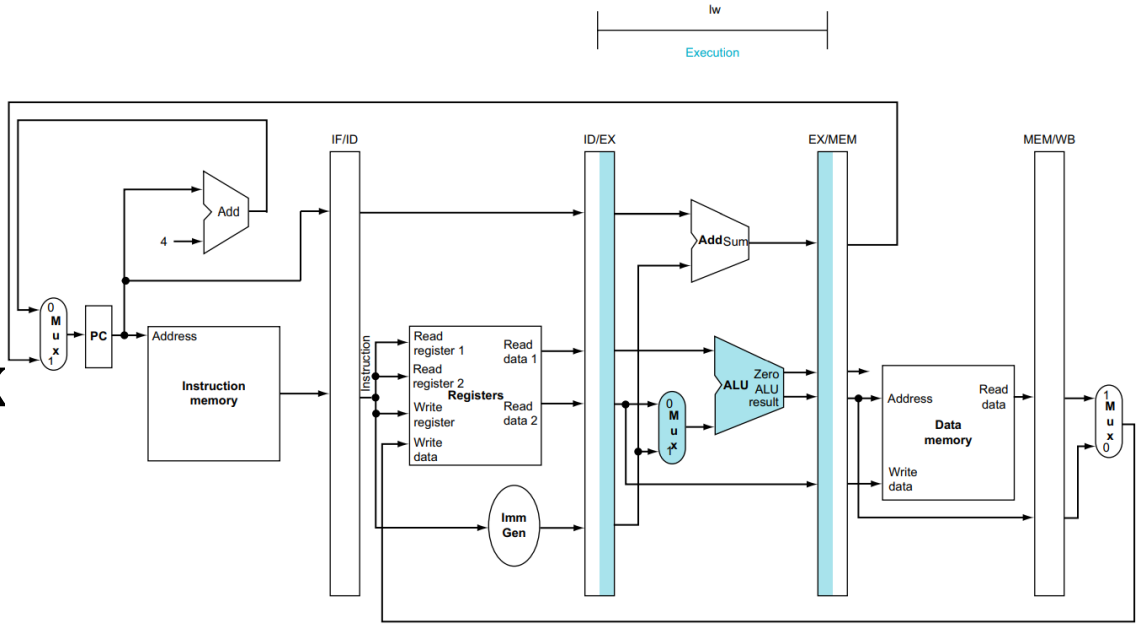


FIGURE 4.39 EX: The third pipe stage of a load instruction, highlighting the portions of the datapath in Figure 4.37 used in this pipe stage. The register is added to the sign-extended immediate, and the sum is placed in the EX/MEM pipeline register

Highlight MEM and WB Stages of Load Inst.

MEM

- The top image shows the load inst. reading the data memory using the address from the EX/MEM pipeline register and loading the data into the MEM/WB pipeline register

WB

- The bottom image shows the final step: reading the data from the MEM/WB pipeline register and writing it into the register file in the middle of the figure

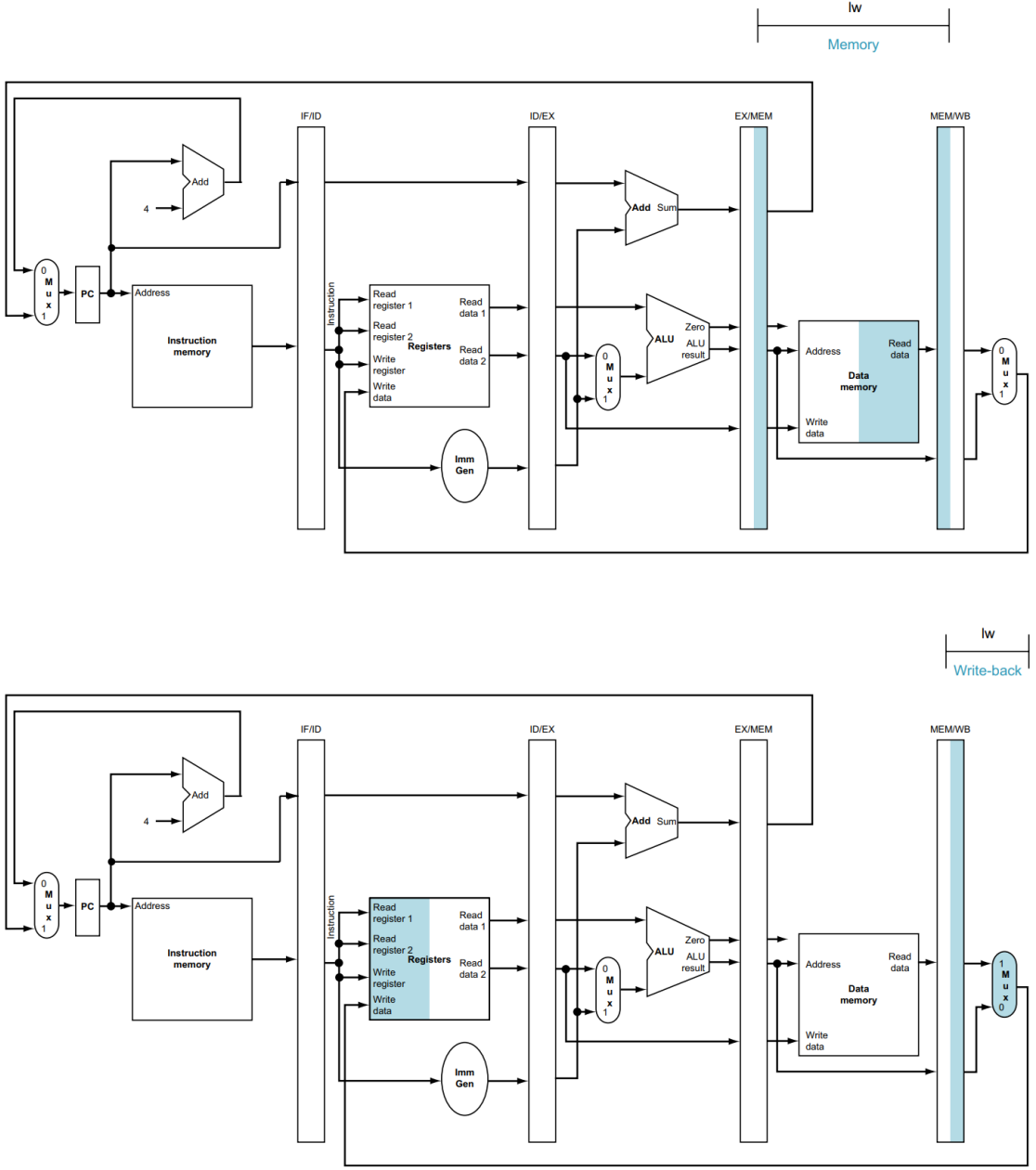
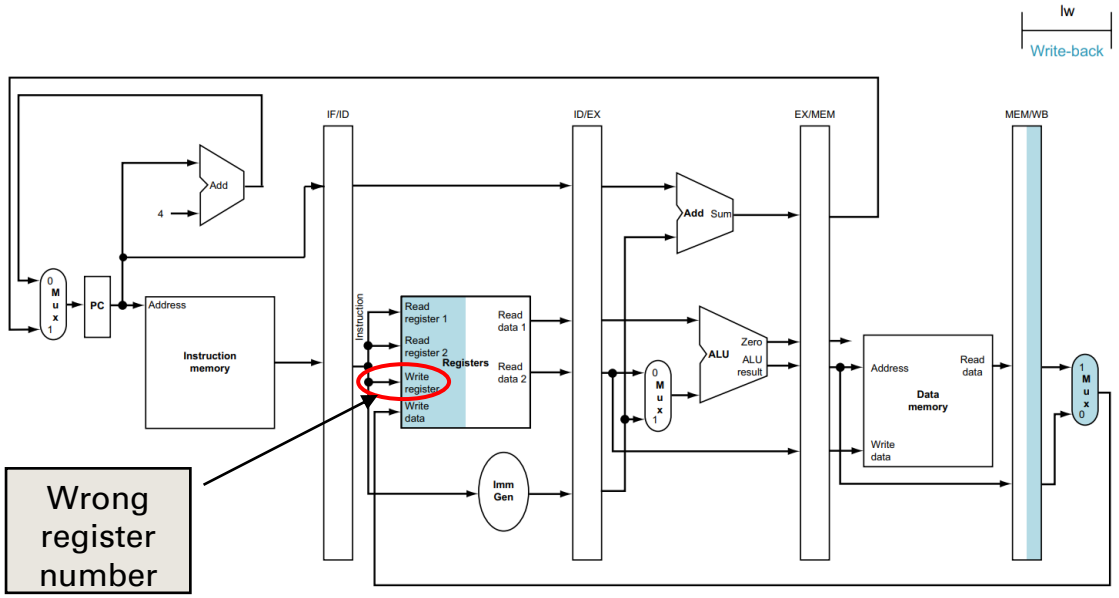


FIGURE 4.40 MEM and WB: The fourth and fifth pipe stages of a load instruction, highlighting the portions of the datapath in Figure 4.37 used in this pipe stage. Data memory is read using the address in the EX/MEM pipeline registers, and the data are placed in the MEM/WB pipeline register. Next, data are read from the MEM/WB pipeline register and written into the register file in the middle of the datapath. Note: there is a bug in this design that is repaired in Figure 4.43

Bug in the WB Stage of Load Inst.

- Bug: Which instruction supplies the write register number?
  - The instruction in the IF/ID pipeline register supplies the write register number, but this instruction occurs considerably after the load instruction
- Solution: Need to preserve the destination register number in the load instruction
  - Load inst. must pass the register number from the ID/EX through EX/MEM to the MEM/WB pipeline register for use in the WB stage
  - Similar mechanism should be applied for store inst.



Bottom of FIGURE 4.40 WB: The fifth pipe stage of a load instruction, highlighting the portions of the datapath in Figure 4.37 used in this pipe stage. Next, data are read from the MEM/WB pipeline register and written into the register file in the middle of the datapath. Note: there is a bug in this design that is repaired in Figure 4.43

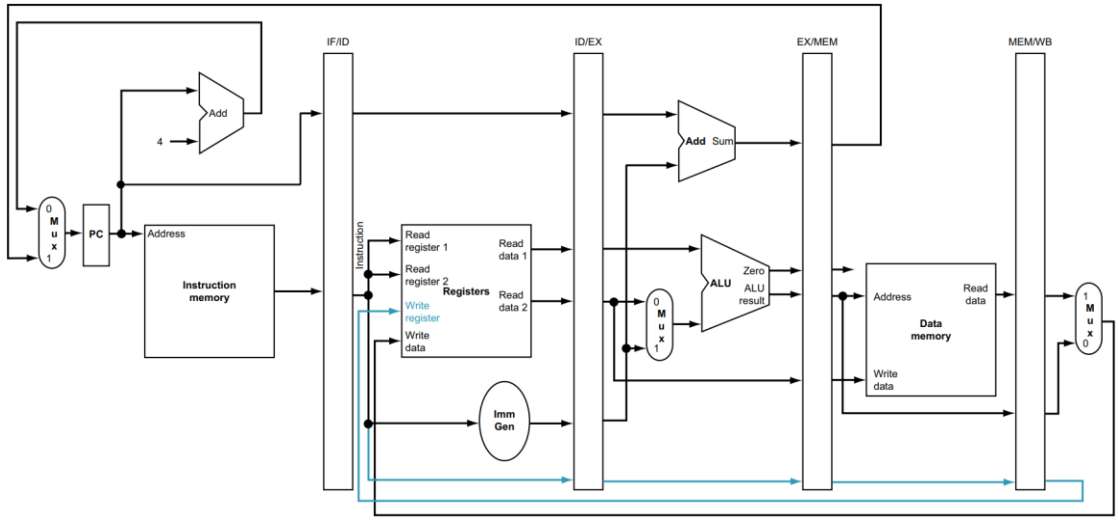


FIGURE 4.43 The corrected pipelined datapath to handle the load instruction properly. The write register number now comes from the MEM/WB pipeline register along with the data. The register number is passed from the ID pipe stage until it reaches the MEM/ WB pipeline register, adding five more bits to the last three pipeline registers. This new path is shown in color

Pipelined datapath (p. 300 ~ 305)

The walk-through of the store inst. in the pipelined datapath

Please read through …

Pipeline Representations

- Two basic styles of pipeline figures
  - to illustrate the concept of simultaneously executing of multiple inst.
  - multiple-clock-cycle pipeline diagrams (Figs. 4.45 (resources used) and 4.46 (stage names))
  - single-clock-cycle pipeline diagrams (Fig. 4.47 (states of the entire datapath))
- Example inst. sequence

lw x10, 40(x1)
sub x11, x2, x3
add x12, x3, x4
lw x13, 48(x1)
add x14, x5, x6

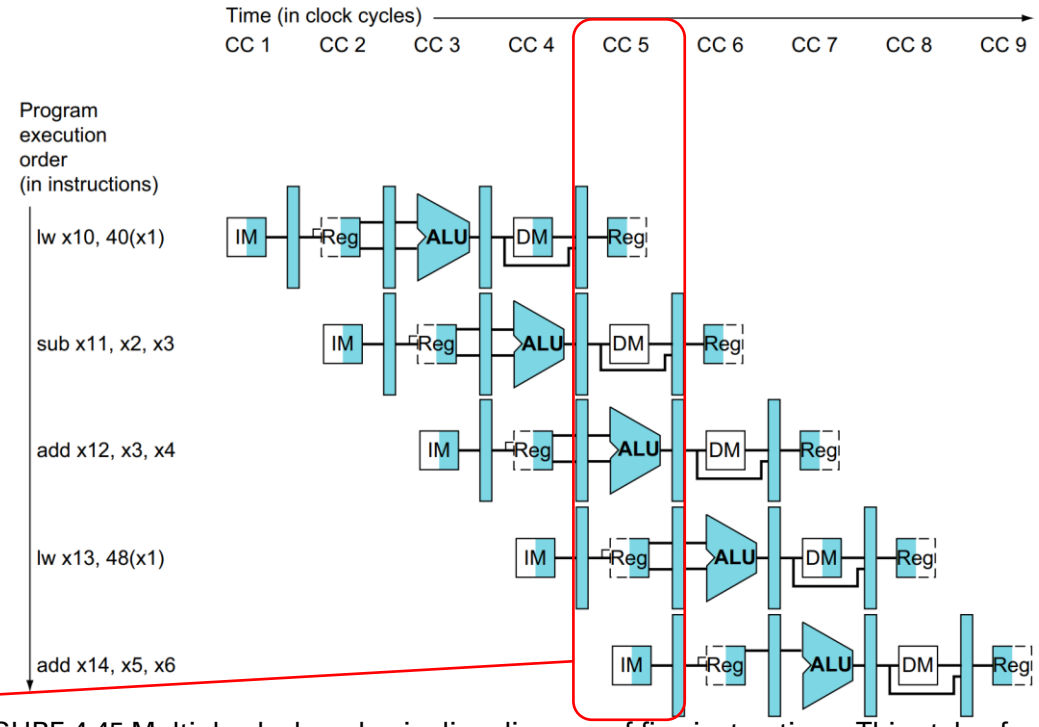


FIGURE 4.45 Multiple-clock-cycle pipeline diagram of five instructions. This style of pipeline representation shows the complete execution of instructions in a single figure. Instructions are listed in instruction execution order from top to bottom, and clock cycles move from left to right. Unlike Figure 4.26, here we show the pipeline registers between each stage. Figure 4.59 shows the traditional way to draw this diagram

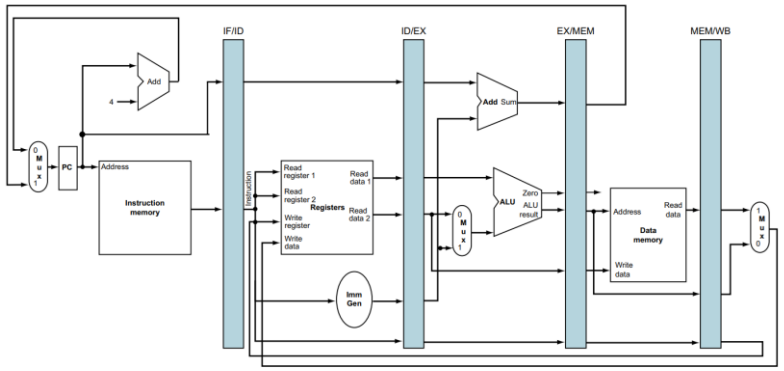


FIGURE 4.47 The single-clock-cycle diagram corresponding to clock cycle 5 of the pipeline in Figures 4.45 and 4.46. As you can see, a single-clock-cycle figure is a vertical slice through a multiple-clock-cycle diagram

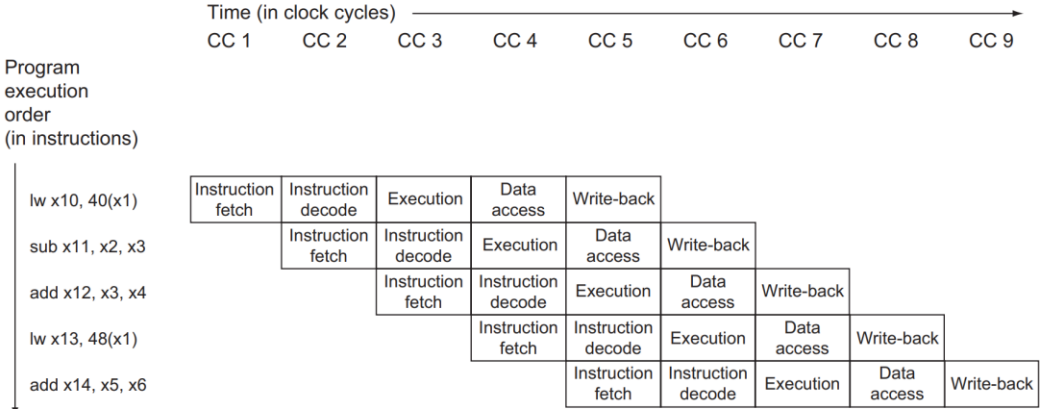


FIGURE 4.46 Traditional multiple-clock-cycle pipeline diagram of five instructions in Figure 4.45

Pipelined Control

- Three steps to add control to the pipelined datapath (as in Sec. 4.4)

1. Label the control lines on the datapath (in Fig. 4.48)
   - ➢Borrow from the control for the simple datapath in Fig. 4.21
   - ➢I.e., we use the same ALU control logic, branch logic, and control lines (Figs. 4.49 ~ 4.51)
2. Determine the units written on each clock cycle for five stages (Fig. 4.51)
   - ➢i.e., PC and the pipeline registers (e.g., IF/ID)
3. Set the control values during each pipeline stage (Figs. 4.52, 4.53)
   - ➢Based on the active component(s) on each stage

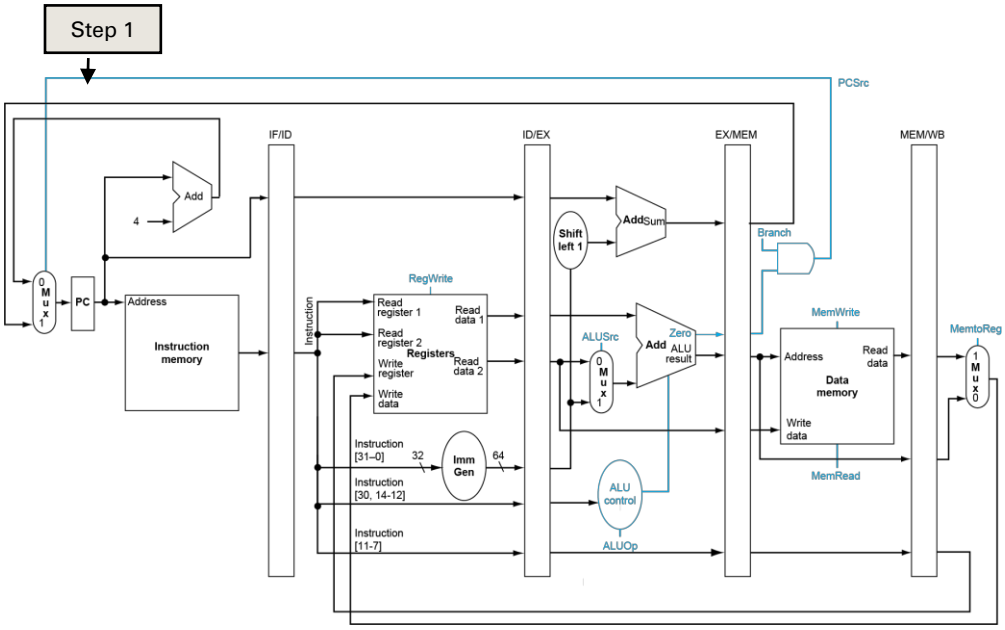


FIGURE 4.48 The pipelined datapath of Figure 4.43 with the control signals identified. This datapath borrows the control logic for PC source, register destination number, and ALU control from Section 4.4. Note that we now need funct fields of the instruction in the EX stage as input to ALU control, so these bits must also be included in the ID/EX pipeline register

Pipelined Control on Five Stages (Step 2)

Nothing to control

Refer to Fig. 4.51

- Instruction fetch
  - The control signals to read instruction memory and to write the PC are always asserted,
  - so there is nothing special to control in this pipeline stage
- Instruction decode/register file read
  - The two source registers are always in the same location in the RISC-V instruction formats,
  - so there is nothing special to control in this pipeline stage
- Execution/address calculation
  - The signals to be set are ALUOp and ALUSrc (see Figs. 4.49 and 4.50)
  - The signals select the ALU operation and either Read data 2 or a sign-extended immediate as inputs to the ALU
- Memory access
  - The control lines set in this stage are Branch, MemRead, and MemWrite
  - The branch if equal, load, and store instructions set these signals, respectively. Recall that PCSrc in Fig. 4.50 selects the next sequential address unless control asserts Branch and the ALU result was 0
- Write-back
  - The two control lines are MemtoReg, which decides between sending the ALU result or the memory value to the register file, and RegWrite, which writes the chosen value

Instruction	ALUOp	operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
lw	00	load word	XXXXXXX	XXX	add	0010
sw	00	store word	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

FIGURE 4.49 A copy of Figure 4.12. This figure shows how the ALU control bits are set depending on the ALUOp control bits and the different opcodes for the R-type instruction

Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

FIGURE 4.50 A copy of Figure 4.20. The function of each of six control signals is defined. The ALU control lines (ALUOp) are defined in the second column of Figure 4.49. When a 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Note that PCSrc is controlled by an AND gate in Figure 4.48. If the Branch signal and the ALU Zero signal are both set, then PCSrc is 1; otherwise, it is 0. Control sets the Branch signal only during a beq instruction; otherwise, PCSrc is set to 0

Since pipelining the datapath leaves the meaning of the control lines unchanged, we can use the same control values
Figure 4.51 has the same values as in Section 4.4, but now the seven control lines are **grouped by pipeline stage**
The seven control lines should be set to these values in each stage for each instruction

Instruction	Execution/address calculation stage control lines		Memory access stage control lines			Write-back stage control lines	
	ALUOp	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	10	0	0	0	0	1	0
lw	00	1	0	1	0	1	1
sw	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

FIGURE 4.51 The values of the control lines are the same as in Figure 4.22, but they have been shuffled into three groups corresponding to the last three pipeline stages

Pipelined Control on Each Stage (Step 3)

- The control lines starts with the EX stage (Fig. 4.52)
  - ➢we create the control information during instruction decode for the later stages
  - ➢Fig. 4.52 shows that these control signals are used in the appropriate pipeline stage as the instruction moves down the pipeline

- Fig. 4.53 shows the full datapath with the extended pipeline registers and with the control lines connected to the proper stage
  - ➢Sec. 4.14 gives more examples of RISC-V code executing on pipelined hardware using single-clock diagrams

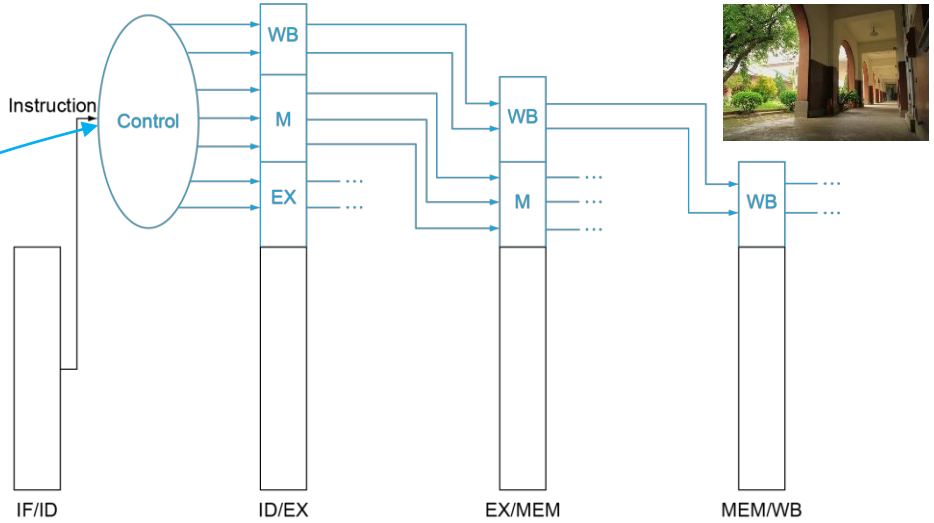


FIGURE 4.52 The seven control lines for the final three stages. Note that two of the seven control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines; three are used during the MEM stage, and the last two are passed to MEM/WB for use in the WB stage

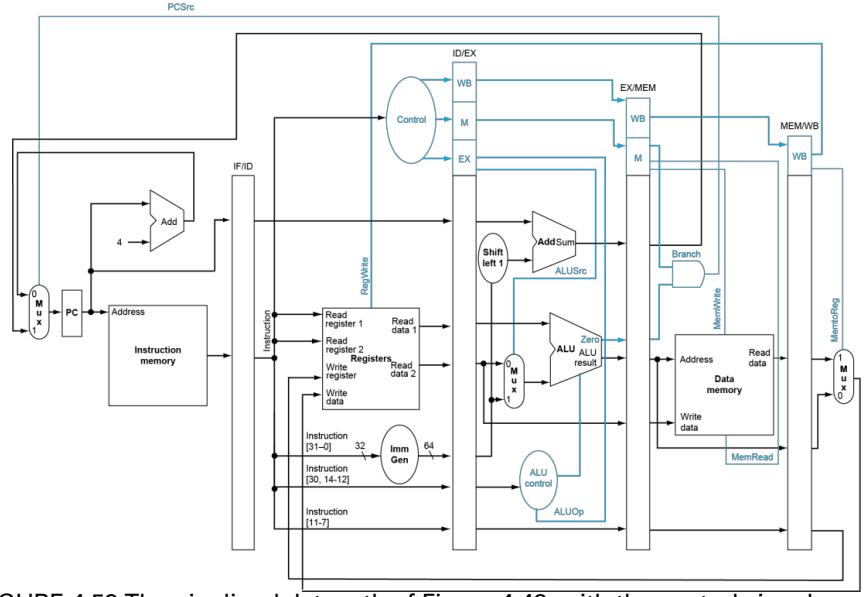
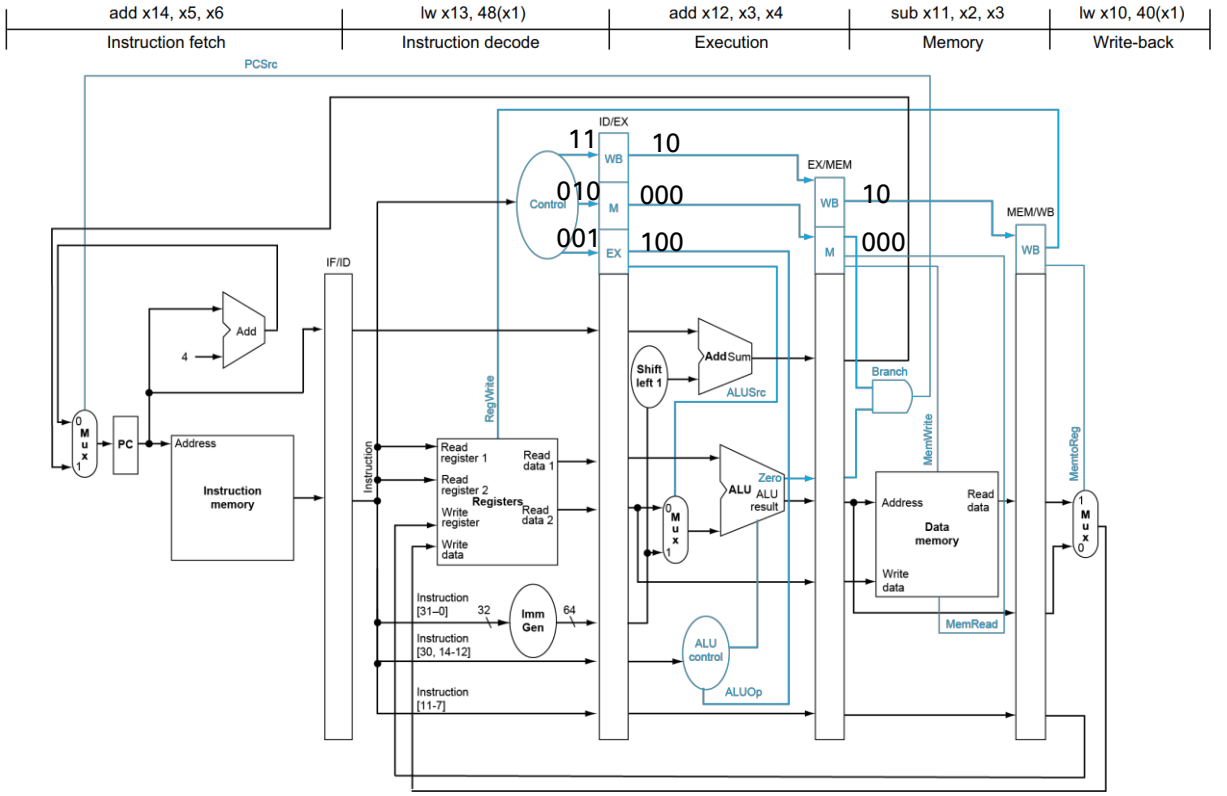


FIGURE 4.53 The pipelined datapath of Figure 4.48, with the control signals connected to the control portions of the pipeline registers. The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage

Pipelined Control on Each Stage (Step 3) A Concrete Example

- The control values on each stage are determined by Fig. 4.51 according to the instruction type

- The figure shows the control values of each stage for the five example instructions



Instruction	Execution/address calculation stage control lines		Memory access stage control lines			Write-back stage control lines	
	ALUOp	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	10	0	0	0	0	1	0
lw	00	1	0	1	0	1	1
sw	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

FIGURE 4.51 The values of the control lines are the same as in Figure 4.22, but they have been shuffled into three groups corresponding to the last three pipeline stages

Data Hazard Part I
Data Hazards in ALU Instructions

- To illustrate the data hazard, we use the code sequence

```
sub  x2,   x1, x3      // Register x2 written by sub
and  x12, x2, x5       // 1st operand(x2) depends on sub
or    x13, x6, x2      // 2nd operand(x2) depends on sub
add  x14, x2, x2       // 1st(x2) & 2nd(x2) depend on sub
sd    x15, 100(x2)     // Base (x2) depends on sub
```

- Many dependencies exist
  - The last four inst. depend on the result in x2 of the first inst.
  - x2 had the value of 10 before sub and -20 afterwards
  - We show how to detect and use forward to resolve the dependencies

Data Hazards in ALU Instructions The Dependencies in Datapath

- A multiple-clock-cycle representation used in Fig. 4.54

- The result of x2 is available during clock cycle 5 (CC 5) or later
  - ➢add and sd can get the correct result
- Incorrect value of x2 since dependency lines go backward
  - ➢and and or inst. get wrong value (10)
  - ➢Solved by data forwarding

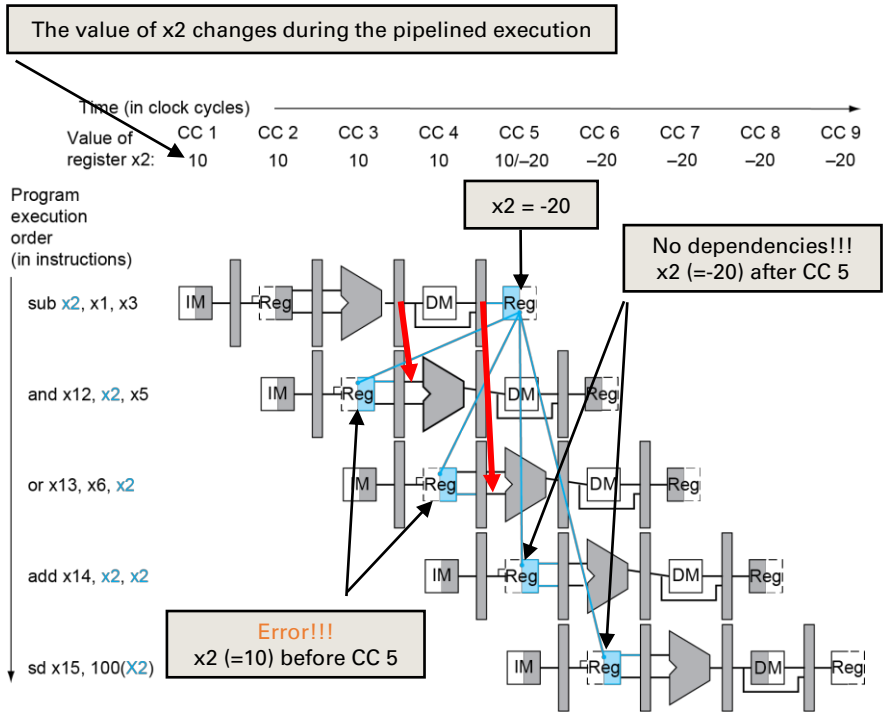


FIGURE 4.54 Pipelined dependences in a five-instruction sequence using simplified datapaths to show the dependences. All the dependent actions are shown in color, and "CC 1" at the top of the figure means clock cycle 1. The first instruction writes into x2, and all the following instructions read x2. This register is written in clock cycle 5, so the proper value is unavailable before clock cycle 5. (A read of a register during a clock cycle returns the value written at the end of the first half of the cycle, when such a write occurs.) The colored lines from the top datapath to the lower ones show the dependences. Those that must go backward in time are pipeline data hazards

Data Hazards in ALU Instructions Notation and Hazards of Pipeline Registers

- Notation for the names the fields of the pipeline registers
  - First part is the name of the pipeline register
  - Second part is the name of the field in this register
  - E.g., ID/EX.RegisterRs1 is register number for Rs1 sitting in ID/EX pipeline register

- This section considers "forwarding to an operation in the EX stage"
  - which may involve with either an ALU operation or an effective address calculation
  - ALU operand register numbers in EX stage are given by ID/EX.RegisterRs1, ID/EX.RegisterRs2

- Two data hazards (dependencies) when

1a. EX/MEM.RegisterRd => ID/EX.RegisterRs1

1b. EX/MEM.RegisterRd => ID/EX.RegisterRs2

2a. MEM/WB.RegisterRd => ID/EX.RegisterRs1

2b. MEM/WB.RegisterRd => ID/EX.RegisterRs2

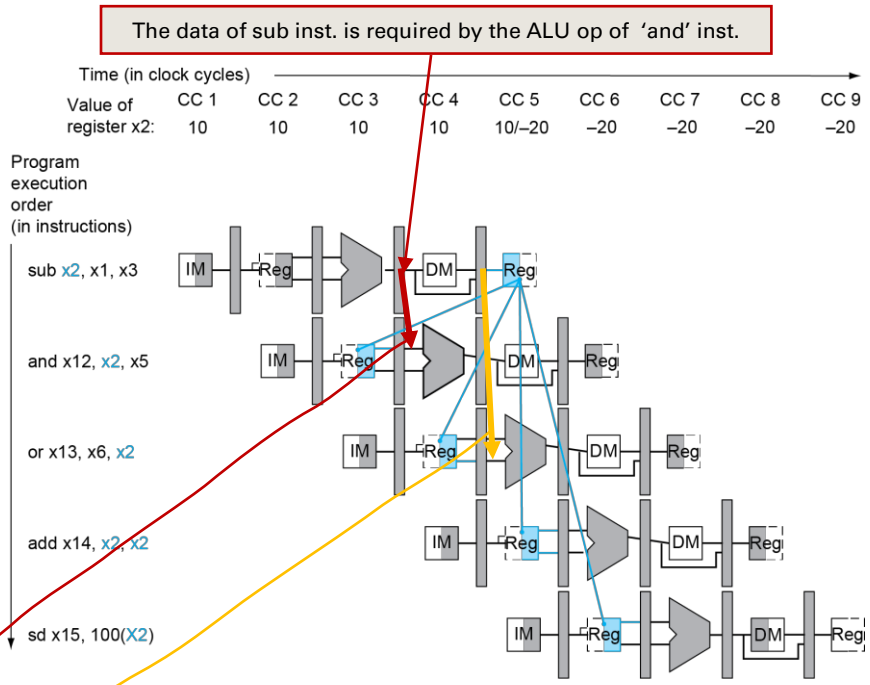
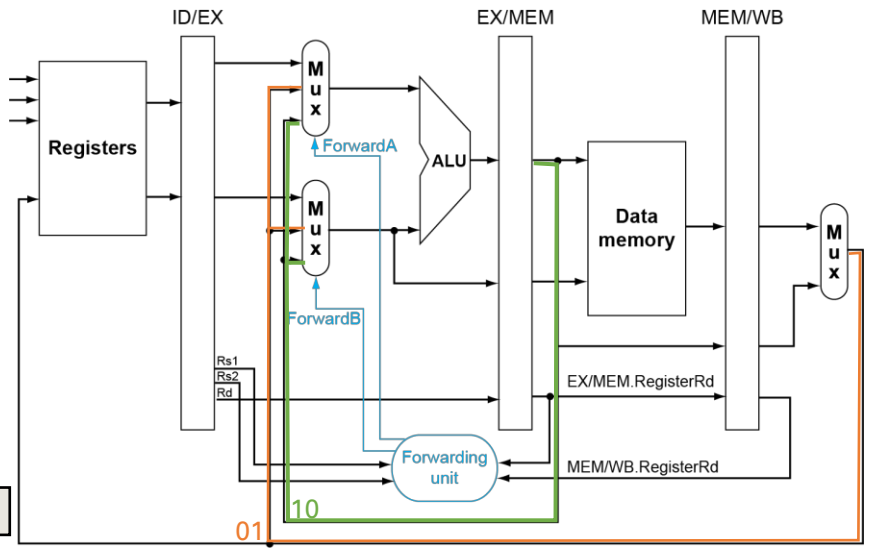


FIGURE 4.54 Pipelined dependences in a five-instruction sequence using simplified datapaths to show the dependences. All the dependent actions are shown in color, and "CC 1" at the top of the figure means clock cycle 1. The first instruction writes into x2, and all the following instructions read x2. This register is written in clock cycle 5, so the proper value is unavailable before clock cycle 5. (A read of a register during a clock cycle returns the value written at the end of the first half of the cycle, when such a write occurs.) The colored lines from the top datapath to the lower ones show the dependences. Those that must go backward in time are pipeline data hazards

Example
- The first hazard in the code sequence on register x2
  - between the result of sub x2, x1, x3 & the first read operand of and x12, x2, x5
- This hazard can be detected when
  the and instruction is in the EX stage and
  the prior instruction is in the MEM stage, so this is hazard 1a

Control Unit to Resolve the Hazards

- Assume the control unit resolves hazards for R-format inst.
  - I.e., add, sub, and, and or
  - The forwarding control will be in the EX stage right before ALU needs the data



b. With forwarding

FIGURE 4.56 On the top are the ALU and pipeline registers before adding forwarding. On the bottom, the multiplexors have been expanded to add the forwarding paths, and we show the forwarding unit. The new hardware is shown in color. This figure is a stylized drawing, however, leaving out details from the full datapath such as the sign extension hardware

- Detect and Resolve hazards
  - EX hazard

Writes to register x0 produces nothing

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
        ForwardA = 10
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
        ForwardB = 10
```

  - MEM hazard

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1))
        ForwardA = 01
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2))
        ForwardB = 01
```

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

FIGURE 4.57 The control values for the forwarding multiplexors in Figure 4.56. The signed immediate that is another input to the ALU is described in the Elaboration at the end of this section.

Resolve Double Data Hazard

- A more complex situation with the following code sequence

```
add x1,x1,x2
add x1,x1,x3
add x1,x1,x4
...
```

- Both (EX and MEM) hazards detected
  - But, we want to use the most recent result only
- Revised MEM hazard
  - Only forward if EX hazard condition isn't true
  - to avoid the concurrent forwarding on both EX and MEM stages

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
        and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1))
        ForwardA = 01
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
        and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2))
        ForwardB = 01
```

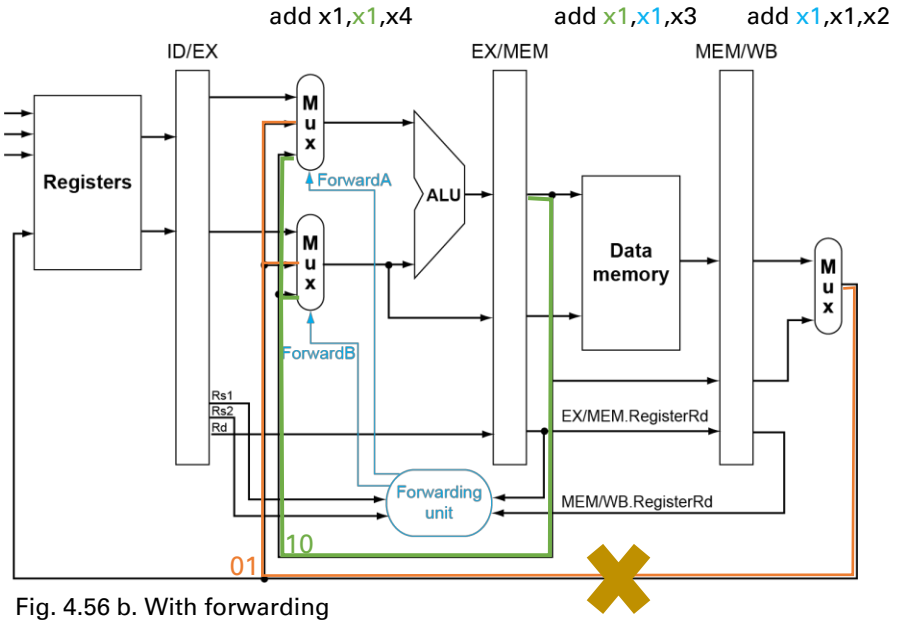


Fig. 4.56 b. With forwarding

The MEM forwarding (of the first add inst.) is not needed when the later instruction (the second add inst.) is going to write the same register, even if the reg. number matches in conditions

Data Hazard Part II
Data Hazard in Load-Use Inst.

- Forwarding does not work
  - ➢when an instruction tries to read a register, and
  - ➢a load instruction that writes the same register
- Fig. 4.60 shows
  - ➢the lw data is still being read from memory in clock cycle 4
  - ➢while the ALU is performing the operation for the following and instruction

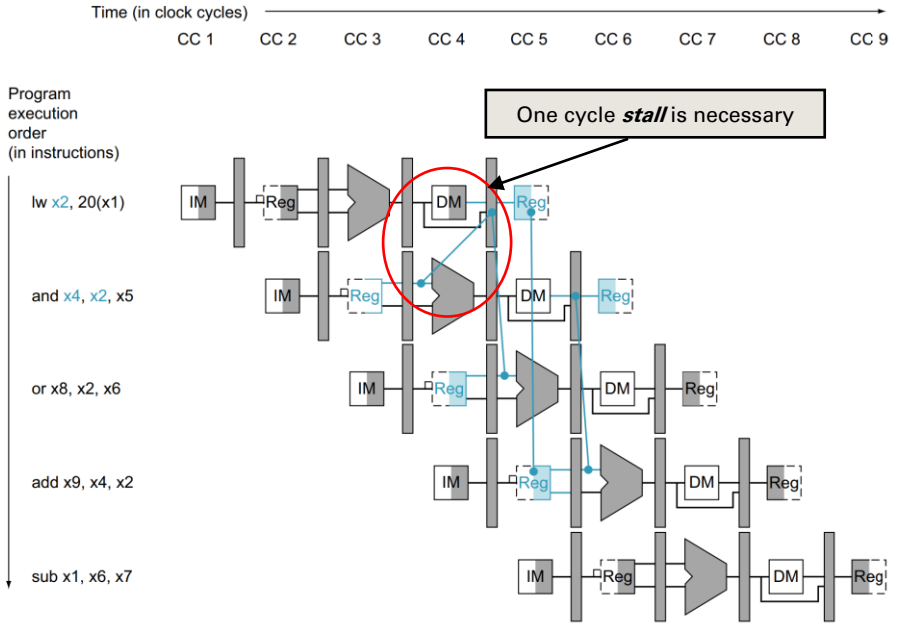


FIGURE 4.60 A pipelined sequence of instructions. Since the dependence between the load and the following instruction (and) goes backward in time, this hazard cannot be solved by forwarding. Hence, this combination must result in a stall by the hazard detection unit

- Stalling for one cycle of the pipeline is needed when
  - ➢the combination of a load followed by an instruction that reads its result

Detecting Data Hazards Caused by Load-Use Inst.

- Use the following condition
  - ➢to detect if it's a load-use hazard, and
  - ➢to stall if a load-use pattern is found

A load inst. in EX stage

The destination reg. of this load inst. matches either source registers of the next inst. in the ID stage

If (ID/EX.MemRead
and ((ID/EX.RegisterRd = IF/ID.RegisterRs1)
or (ID/EX.RegisterRd = IF/ID.RegisterRs1))
stall the pipeline

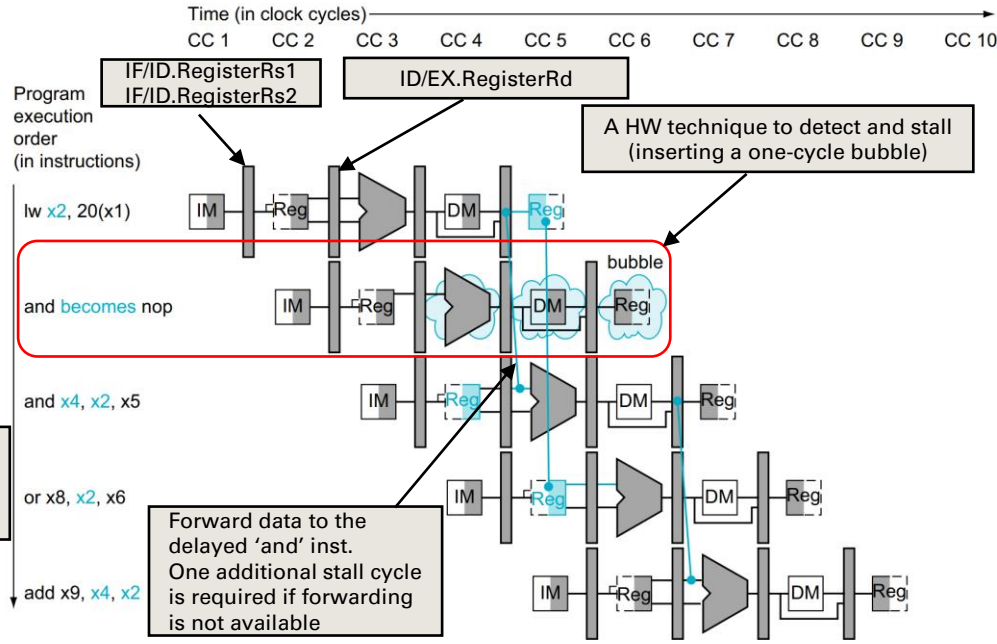


FIGURE 4.61 The way stalls are really inserted into the pipeline. A bubble is inserted beginning in clock cycle 4, by changing the and instruction to a nop. Note that the and instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle 5 (versus the unstalled position in clock cycle 4). Likewise, the or instruction is fetched in clock cycle 3, but its ID stage is delayed until clock cycle 5 (versus the unstalled clock cycle 4 position). After insertion of the bubble, all the dependences go forward in time and no further hazards occur

- ➢It is a HW-based solution
- ➢A SW solution is to insert NOP instructions
  - ❖NOP: An instruction that does no operation to change state

Stall the Pipeline to Handle the Load-Use Hazard

- If the instruction in the ID stage is stalled,
  - ➢then the instruction in the IF stage must also be stalled too
- The stall is done by
  1. preventing the PC register and the IF/ID pipeline register from changing (disabling the writes to PC and IF/ID pipeline register)
  2. setting all seven control signals to zero in EX, MEM, and WB stages

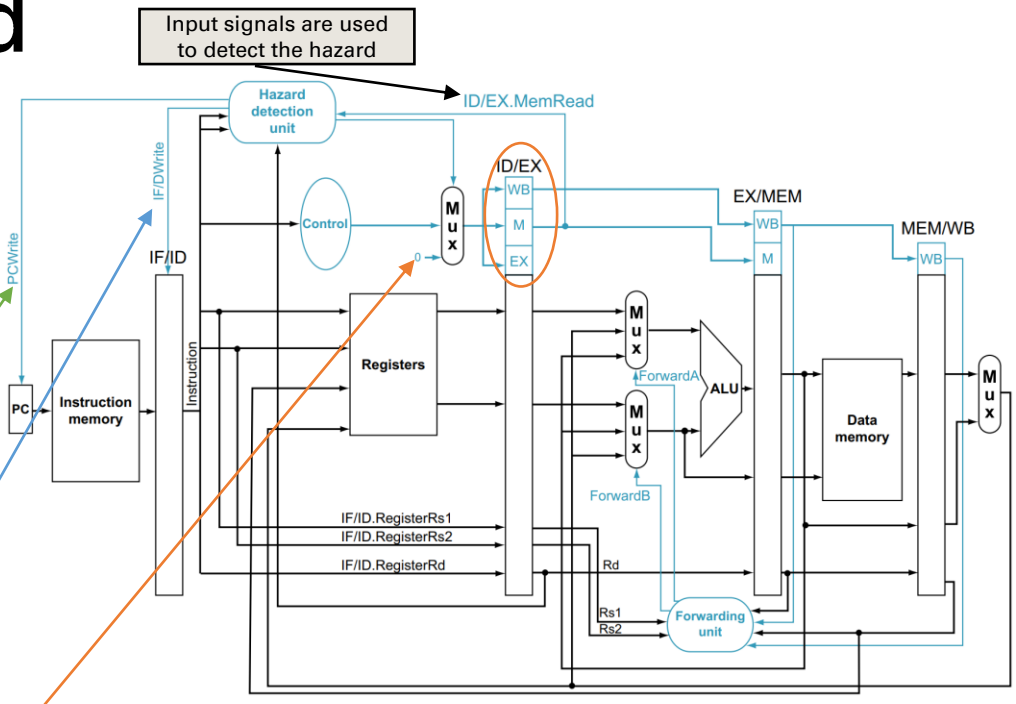


FIGURE 4.62 Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and the forwarding unit. Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing— this drawing gives the essence of the forwarding hardware requirements

Instruction	Execution/address calculation stage control lines		Memory access stage control lines			Write-back stage control lines	
	ALUOp	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	10	0	0	0	0	1	0
lw	00	1	0	1	0	1	1
sw	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

FIGURE 4.51.
Set the seven control lines to zero for the pipeline to ***stall***

Control Hazards

- Stalling until the branch is determined is too slow
  - Recap: Fig. 4.33 assumes *taken* w/ *forwarding*
- Branch decision is determined in MEM stage
  - Control hazard occurs and it wastes three cycles in the worst case scenario

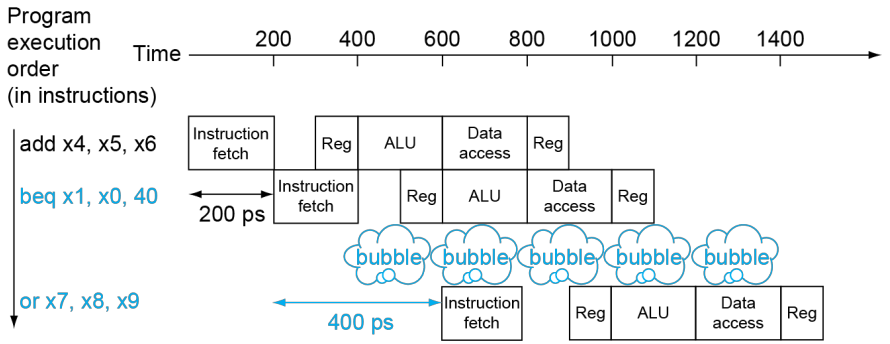
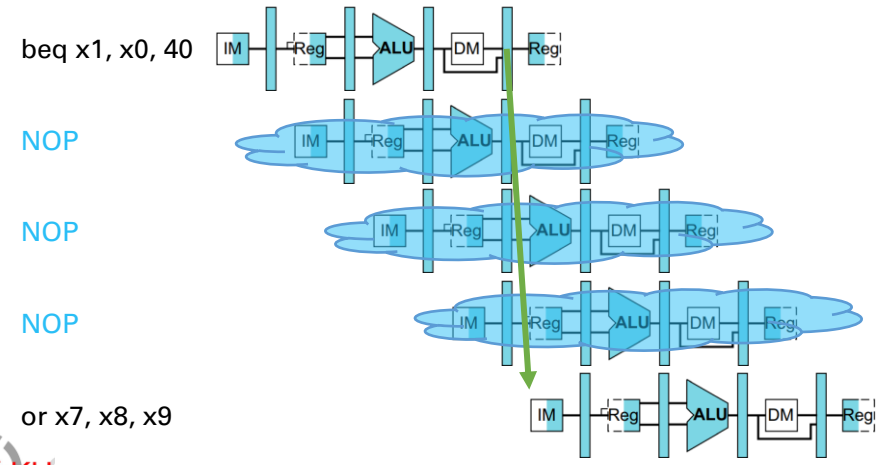


FIGURE 4.33 Pipeline showing stalling on every conditional branch as solution to control hazards. This example assumes the conditional branch is taken, and the instruction at the destination of the branch is the "or" instruction. There is a one-stage pipeline stall, or bubble, after the branch. In reality, the process of creating a stall is slightly more complicated, as we will see in Section 4.9. The effect on performance, however, is the same as would occur if a bubble were inserted



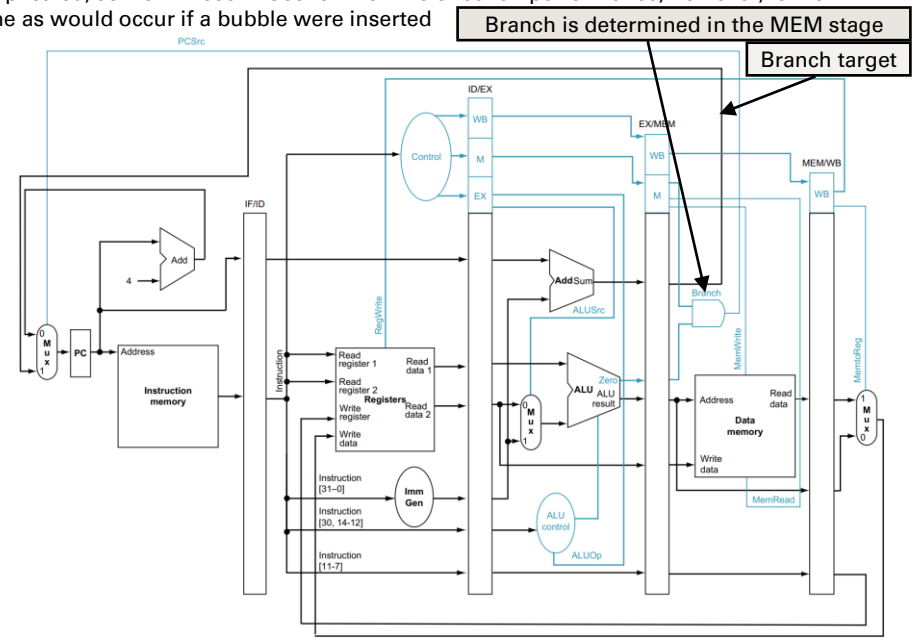
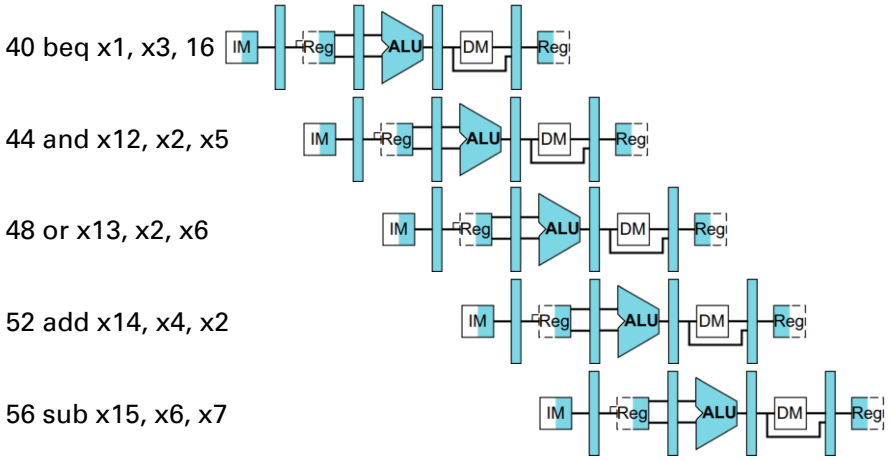


FIGURE 4.53 The pipelined datapath of Figure 4.48, with the control signals connected to the control portions of the pipeline registers. The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage

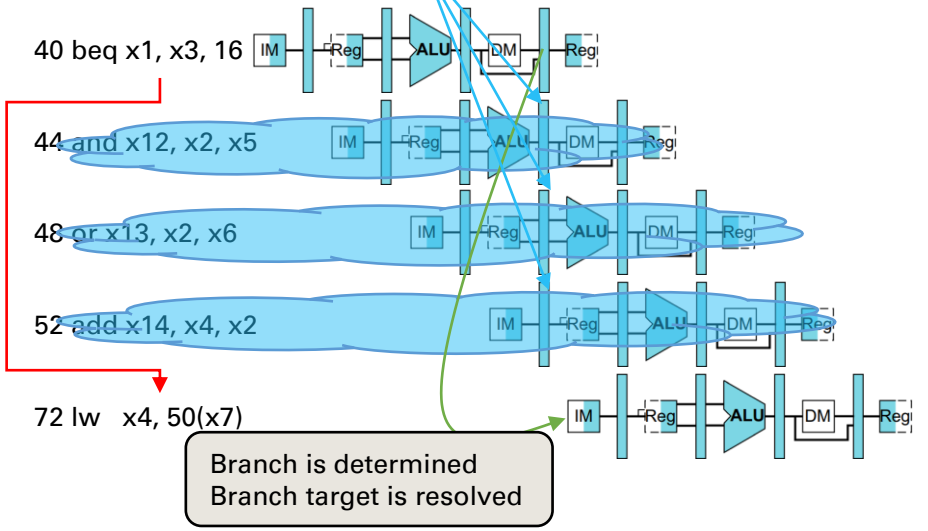
Control Hazards (Sol 1: Prediction)

- Predict not-taken conditional branches
  - ➢Continue execution down the sequential instruction stream
  - ➢Correct prediction has no penalty
  - ➢Wrong prediction leads to the pipeline flush
    - ❖I.e., discard (flush) instructions in the IF, ID, and EX stages of the pipeline (clear their pipeline registers) when the branch reaches the MEM stage

Assume non-taken branches
Correct prediction
Normal execution



Assume non-taken branches
Wrong prediction
Needs to *flush* the pipeline



Control Hazards (Sol 2: Reduce Branch Delay)

- Move the conditional branch execution earlier
  - The next PC of a branch is determined in MEM stage (flush 3 inst.)
  - If the next PC of a branch is determined earlier, fewer inst. will be flushed
- Two things needed to be done earlier in ID stage

1. Calculate branch target address
   - Move the branch adder from the EX stage to the ID stage
   - Simple; The PC value and the immediate field are in the IF/ID pipeline register
2. Evaluate branch decision

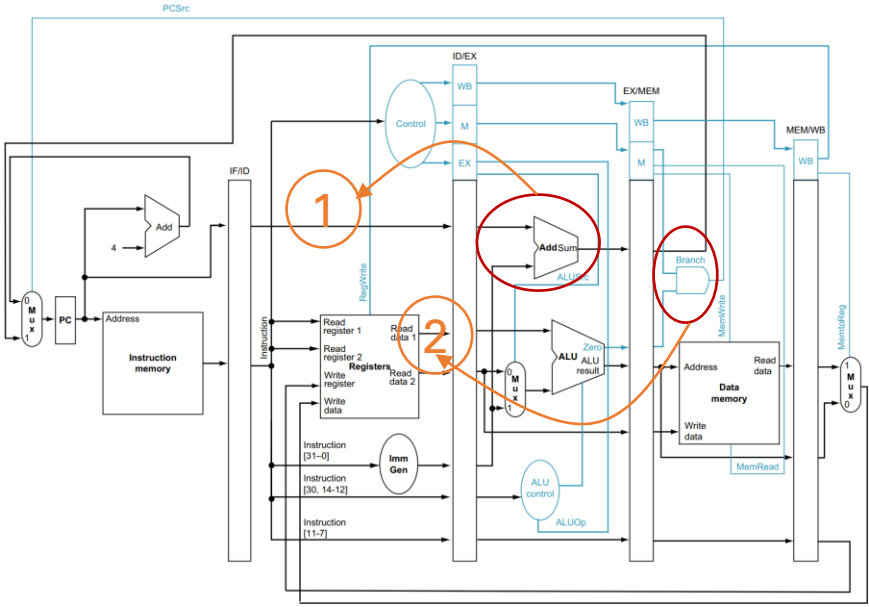


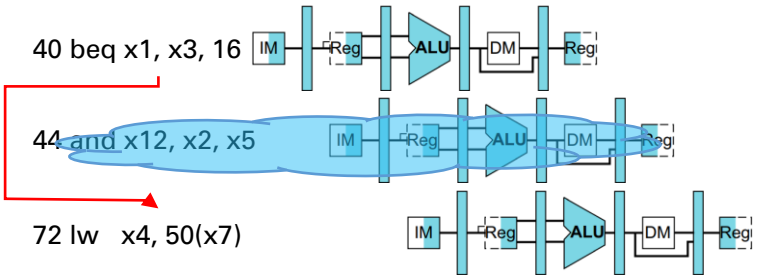
FIGURE 4.53

Control Hazards (Sol 2: Reduce Branch Delay; Cont'd)

- Move the conditional branch execution earlier
- Two things needed to be done earlier in ID stage

1. Calculate branch target address
2. Evaluate branch decision
   - ➢ For branch if equal, equality test can be tested
     - ❖ by XORing individual bit positions of two registers and ORing the XORed result (A zero output of the OR gate means the two registers are equal)
   - ➢ Additional forwarding and hazard detection hardware are needed (check textbook p. 327)
     - ❖ since a branch dependent on a result still in the pipeline (e.g., branch if equal) must still work properly with this optimization

- This optimization reduces the penalty of a branch to only one instruction if the branch is taken (i.e., the one currently being fetched, e.g., the and inst. at addr. of 44; it wastes of one cycle instead of three cycles in Sol 1)
- To flush instructions in the IF stage, we add a control line (IF.Flush) that zeros the instruction field of the IF/ID pipeline register
- Clearing the register transforms the fetched instruction into a NOP, an instruction that has no action and changes no state



A simplified illustration; a detailed version is in the next page

Control Hazards (Sol 2: Reduce Branch Delay; Cont'd)

- Assume the always-no-taken branches w/ the code seq

40 beq x1, x3, 16

44 and x12, x2, x5

48 or x13, x2, x6

52 add x14, x4, x2

56 sub x15, x6, x7

...

72 lw x4, 50(x7)

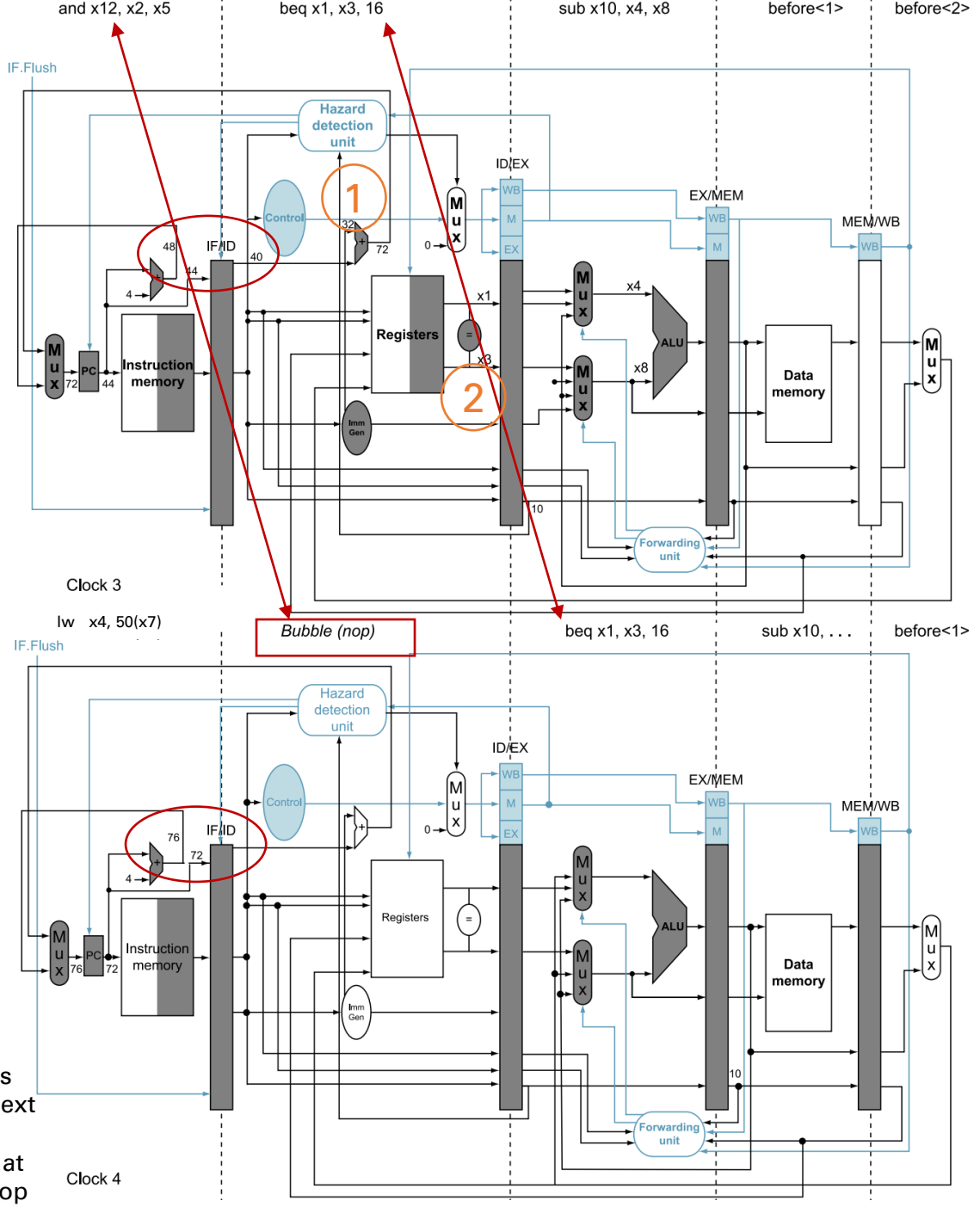


FIGURE 4.64 The ID stage of clock cycle 3 determines that a branch must be taken, so it selects 72 as the next PC address and zeros the instruction fetched for the next clock cycle. Clock cycle 4 shows the instruction at location 72 being fetched and the single bubble or nop instruction in the pipeline because of the taken branch.

Control Hazards (Sol 3: Dynamic Branch Prediction)

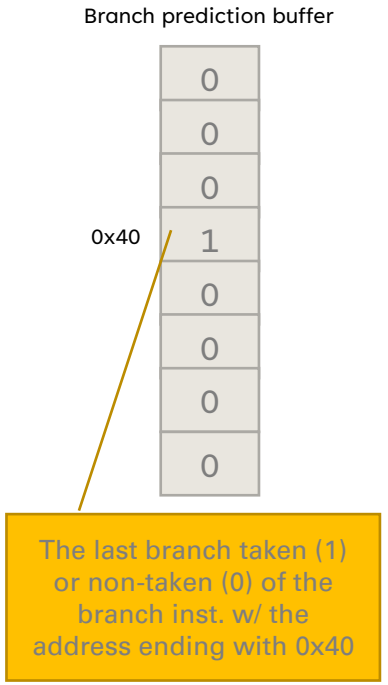
- Static branch prediction
  - ➢is sufficient for a simple pipeline design (the five-stage pipeline)
  - ➢As the branch penalty increases (inst. Lost) in deeper and superscalar pipelines, the static approach wastes too much performance

- Dynamic branch prediction
  - ➢Predict branch behavior during program execution based on the history of branches
  - ➢To look up the address of the instruction to see if the conditional branch was taken the last time for this instruction was executed, and,
  - ➢if so, to begin fetching new instructions from the same place as the last time

Control Hazards
(Sol 3: Dynamic Branch Prediction Impl.)

- Implemented w/ branch prediction buffer (or branch history table)
  - ➢Is a small memory array indexed by the lower portion of the addresses of the recent branch instructions when the inst. are fetched
  - ➢Each array entry stores whether the branch was recently taken or not

- The use of the branch prediction buffer
  - ➢When a branch is executed, the table is check to help predict the branch result
  - ➢to fetch from the fall-through (next PC of the branch inst.) or the target address
  - ➢The example in the right is a 1-bit predictor
    - ❖ Initialized to zeros for all entries
    - ❖ Use the history (what's on the table) to predict this branch result
    - ❖ Update the entry with the branch result
    - ❖ What is the *accuracy* of the four branches: non-taken, non-taken, non-taken, taken?



Branch sequence	N, N, N, T
Prediction history	0, 0, 0, 0
Predicted branch	N, N, N, N
Correct or not?	Y, Y, Y, N

Control Hazards (Sol 3: 2-bit Branch Predictor)

- In a 2-bit scheme, a prediction must be wrong twice (two successive mispredictions) before it is changed

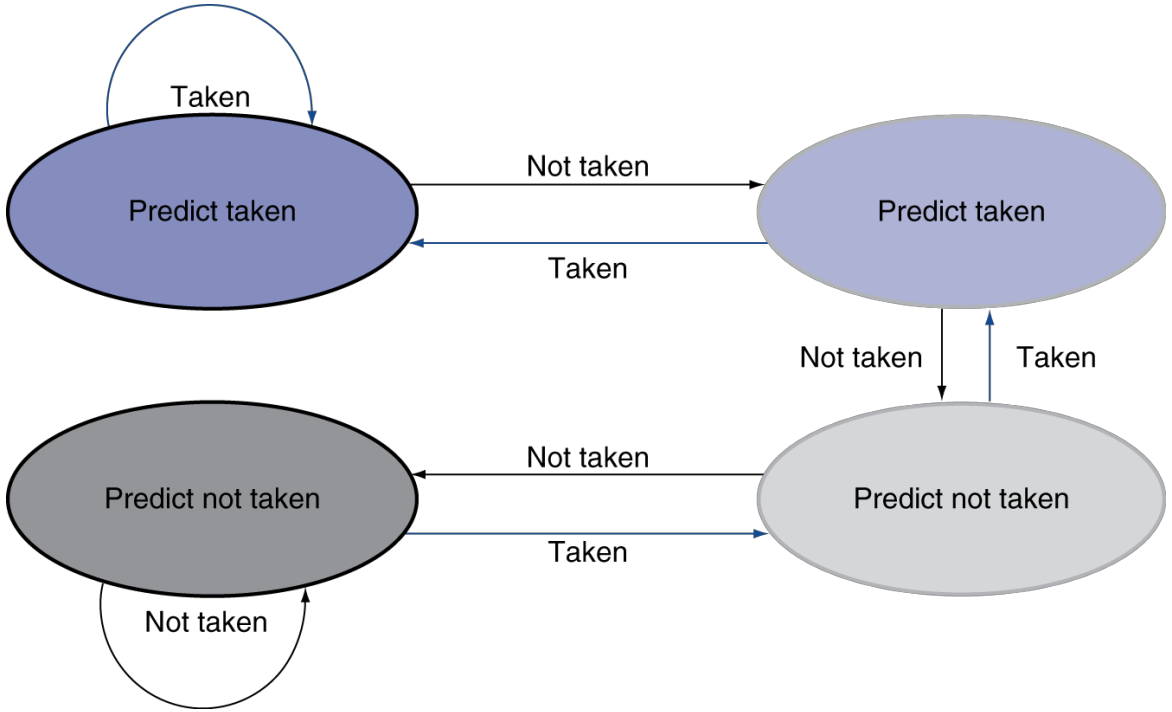


FIGURE 4.65 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted only once. The 2 bits are used to encode the four states in the system. The 2-bit scheme is a general instance of a counter-based predictor, which is incremented when the prediction is accurate and decremented otherwise, and uses the mid-point of its range as the division between taken and not taken

Please read through …

The example in p. 330

The Loops and Prediction example in Sec. 4.9

Variants of branch predictor in p. 331

Correlating predictor

Tournament branch predictor

Elaborations

"Check Yourself" in p. 332

To calculate the accuracy of different types of branch predictor.
It is at the bottom of page 332

Summary of control and data hazards

Recap:
Data hazards caused by arithmetic ops, data transfer ops
Control hazards caused by control instructions

Solution:
Data hazard overhead alleviated by code reordering (solved partially by forwarding)
Control hazard overhead alleviated by code reordering and branch prediction

Exceptions

- "Unexpected" events requiring change in flow of control
  - Events other than branches that change the normal flow of instruction execution
  - Dealing with them without sacrificing performance is hard
- These events are called exceptions or interrupts
  - Different ISAs use the terms with different meanings
- Exception: any unexpected change in control flow without distinguishing whether the cause is internal or external
  - Arises within the CPU
  - E.g., undefined opcode, syscall, overflow of an operation, ···
- Interrupt: the event is externally caused
  - From an external I/O device (controller), or user (system reset)

Type of event	From where?	RISC-V terminology
System reset	External	Exception
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Either

Exception Handling

- When an exception occurs (e.g., during the exe. of the add inst.), the actions are taken

① ➢ The processor saves the address of the unfortunate instruction (the interrupted inst.) in the RISC-V 64-bit supervisor exception cause register (SEPC)

② ➢ then transfers control to OS at some specified address (e.g., the single-entry exception lookup routine) to look up the RISC-V 64-bit Supervisor Exception Cause Register (SCAUSE) for the reason of the exception by decoding SCAUSE content

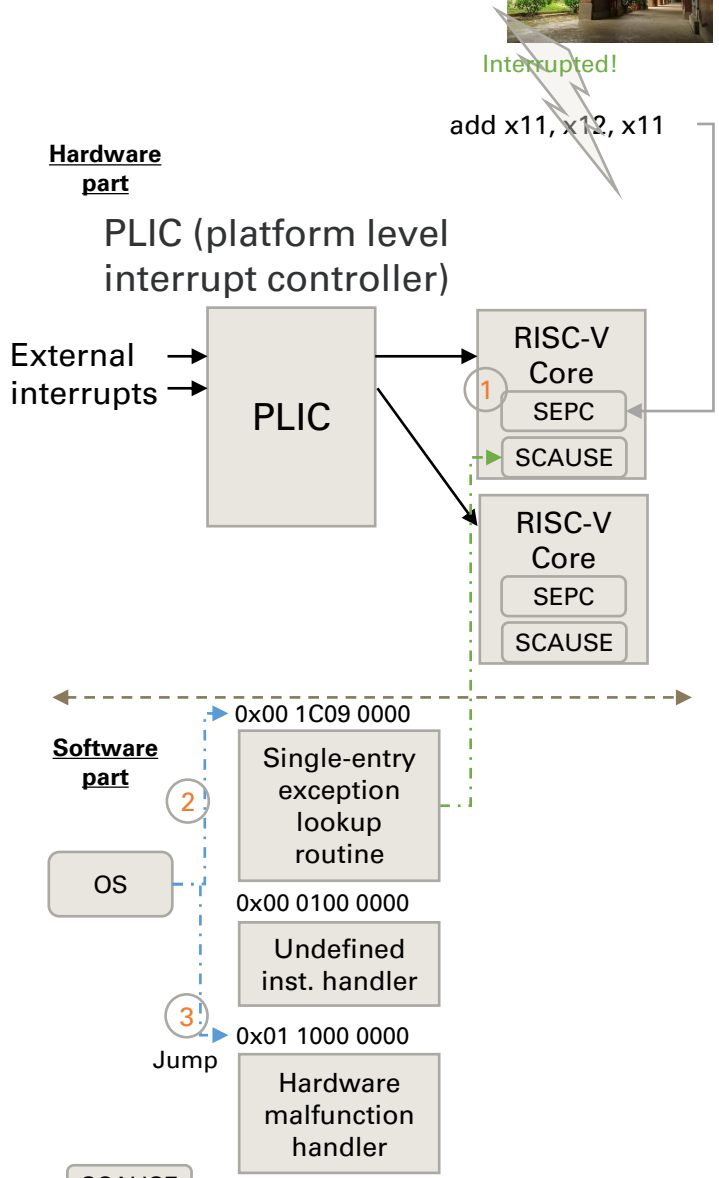
- ❖ E.g., providing some service to the user program (syscall),or taking some predefined action in response to a malfunction

③ ➢ OS jumps to the specific event handler

- ❖ based on the decoding result (check the table below for the mapping of the event and the address of the event handler)

➢ After performing whatever action (exception handling) is required, OS may continue the execution of the interrupted program

➢ NOTE: Vectored Interrupt is another method used by MIPS and ARM systems to handle interrupts, where the address to which control is transferred is determined by the cause of the exception



SCAUSE

Exception type	Exception vector address to be added to a Vector Table Base Register
Undefined instruction	$00\ 0100\ 0000_{two}$
System Error (hardware malfunction)	$01\ 1000\ 0000_{two}$

STVEC 0x00 1C09 0000

Implement Exceptions in Pipeline

- Exception is treated as another form of control hazard
  - Similar to handling of the branch inst., we must
  - flush the instructions that follow the interrupted instruction (e.g., the add inst.) from the pipeline and
  - fetch instructions from the new address
- A problem with exception handling
  - if we do not stop execution in the middle of the instruction, the programmer will not be able to see the original value of register x11 because it will be clobbered as the destination register of the add instruction
  - The easiest way to do this is to flush the instruction and restart it from the beginning after the exception is handled
- For example, if we assume the exception is detected during the EX stage of the add inst.,
  - we can use the EX.Flush signal to prevent the instruction in the EX stage from writing its result in the WB stage

- Some controls needs to be added (Fig. 4.67)
- To flush inst. in IF stage,
  - as we did for branch misprediction handling, we turn it into a nop with IF.Flush
- To flush inst. in ID stage,
  - we use the multiplexor already in the ID stage that zeros control signals for stalls
  - A new control signal, ID.Flush, is ORed with the stall signal from the hazard detection unit to flush during ID
- To flush inst. in EX phase,
  - we use a new signal called EX.Flush to cause new multiplexors to zero the control lines.
- BTW, the address of the single-entry exception look-up routine is fed to the PC Mux in the left of the fig below

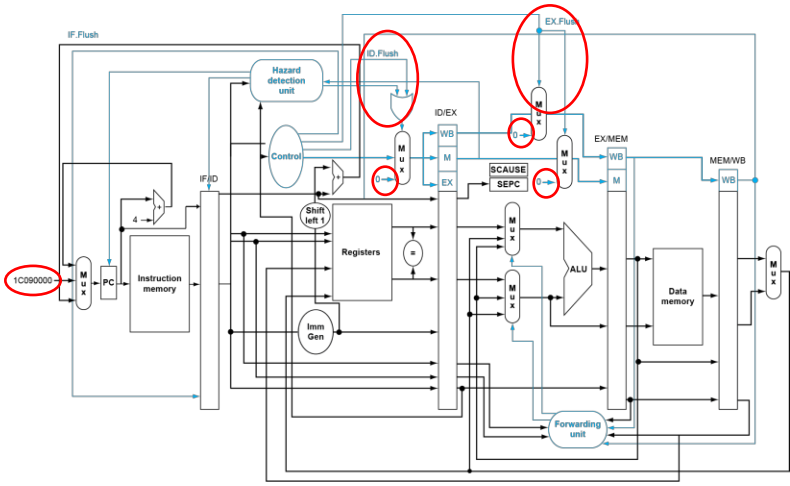


FIGURE 4.67 The datapath with controls to handle exceptions. The key additions include a new input with the value 0000 0000 1C09 0000hex in the multiplexor that supplies the new PC value; an SCAUSE register to record the cause of the exception; and an SEPC register to save the address of the instruction that caused the exception. The 0000 0000 1C09 0000hex input to the multiplexor is the initial address to begin fetching instructions in the event of an exception

An Example of Exception Handling

- Given this instruction sequence,

  40 sub x11, x2, x4

  44 and x12, x2, x5

  48 or x13, x2, x6

  4C add x1, x2, x1

  50 sub x15, x6, x7

  54 lw x16, 100(x7) ...

HW malfunction exception raises →

- assume the instructions to be invoked on an exception begin like this (single-entry exception lookup routine):

  1C09 0000 sw x26, 1000(x10)

  1C09 0004 sw x27, 1008(x10) ...

- Fig. 4.68 shows what happens in the pipeline
  - 1C09 0000 is fed to PC at clock 6
  - 4C is saved in SEPC at clock 6
  - Inst. before add are flushed at clock 7 (there are bubbles)
  - Single-entry exception lookup routine is executed at clock 7
  - The previous inst. are executed normally (i.e., 48 or and 44 and)

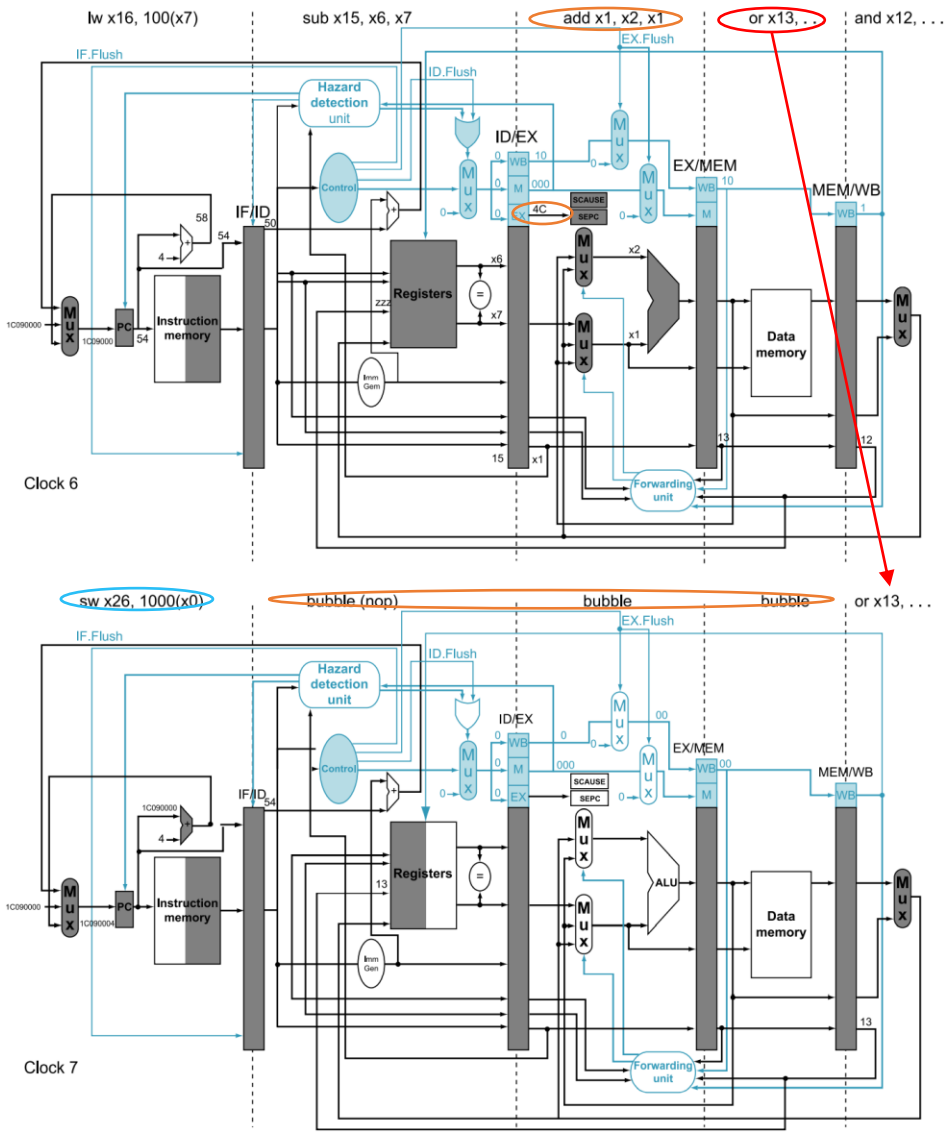


FIGURE 4.68 The result of an exception due to hardware malfunction in the add instruction. The exception is detected during the EX stage of clock 6, saving the address of the add instruction in the SEPC register (4Chex). It causes all the Flush signals to be set near the end of this clock cycle, deasserting control values (setting them to 0) for the add. Clock cycle 7 shows the instructions converted to bubbles in the pipeline plus the fetching of the first instruction of the exception routine—sw x26, 1000(x0)—from instruction location 0000 0000 1C09 0000hex. Note that the and and or instructions, which are prior to the add, still complete

Instruction Level Parallelism

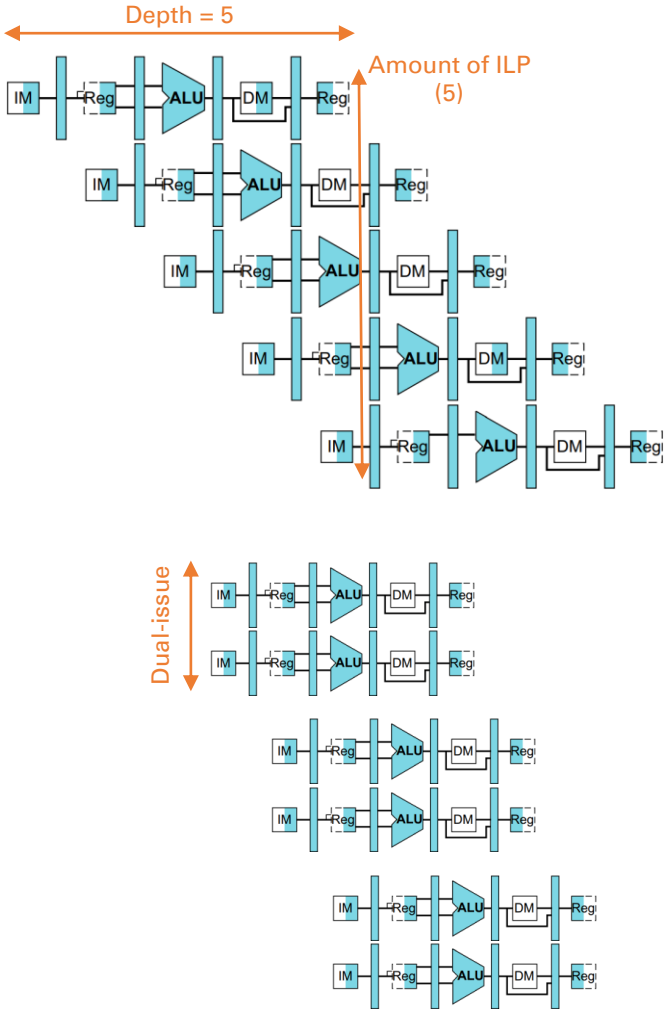
- Pipelining exploits parallelism among inst.
  - by executing multiple instructions in parallel
  - This parallelism is called *instruction-level parallelism* (ILP)
  - This section is covered in more detailed in our graduate class (textbook: Computer Architecture: A Quantitative Approach)

Two methods increase ILP:

1. Deeper pipeline: Increase the depth of the pipeline (overlapping more instructions)
   - Less work per stage => shorter clock cycle
   - Hard to be done now → Hit the physical limits

2. Multiple issue: Replicate internal components of the computer to launch multiple inst. in every pipeline stage
   - Require extra work to keep all the machines busy and
   - transferring the loads to the next pipeline stage

- Multiple instructions per cycle leads to CPI < 1 → use IPC instead
  - E.g., 4GHz 4-way multiple-issue processor
  - 16 billion inst./sec, peak CPI = 0.25, peak IPC = 4
  - But dependencies reduce this in practice



Multiple Issue Processor

- Two main ways to implement a multi-issue processor
  - The division of work between compiler and hardware

- Static multiple issue (compiler-based solution)
  - Compiler groups instructions to be issued together
  - Packages them into "issue slots"
  - Compiler detects and avoids hazards

- Dynamic multiple issue (hardware-based solution)
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

Primary and distinct responsibilities in a multi-issue processor

1. Packaging instructions into issue slots:
   how does the processor determine **how many instructions and which instructions** can be issued in a given clock cycle? In most static issue processors, this process is at least partially handled by the compiler; in dynamic issue designs, it is normally dealt with at runtime by the processor, although the compiler will often have already tried to help improve the issue rate by placing the instructions in a beneficial order
2. Dealing with data and control hazards:
   in static issue processors, the compiler handles some or all the consequences of data and control hazards statically. In contrast, most dynamic issue processors attempt to alleviate at least some classes of hazards using hardware techniques operating at execution time

Speculation

- Speculation, the great idea of prediction,
  - is an approach that allows the compiler or the processor to "guess" about the properties of an instruction
  - to enable execution to begin ASAP for other instructions that may depend on the speculated instruction
  - "Guess" is correct, complete the operation
  - "Guess" is wrong, roll back and do the right thing
  - E.g., branch prediction and precedes a load beforehand

- Speculation could be done via compiler or HW
  - Compiler/Hardware can reorder the execution order of inst.
  - e.g., move an inst. before branch
  - Can include "fix-up" instructions to recover from incorrect guess

- Recovery mechanisms are different on SW and HW
  - Compiler inserts additional inst. to check for wrong speculation
  - Hardware buffers the speculative results until the result is available
    - Speculation correct → write the contents in buffers to reg./mem.
    - Speculation incorrect → flush buffers and re-execute the correct inst. sequence

Static Multiple Issue

- Compiler is responsible for
  - grouping instructions into “issue packets”
  - handling hazards

- In a static issue processor, a group of instructions that can be issued on a single cycle are called an *issue packet*
  - A static multiple-issue processor usually restricts what mix of instructions can be initiated in a given clock cycle
  - It is useful to think of the issue packet as a single instruction allowing several operations in certain predefined fields
  - The operations are determined by available pipeline resources
  - Think of an issue packet as a very long instruction operates with multiple concurrent operations
  - The original name of this approach is Very Long Instruction Word (VLIW)

- Compiler takes some responsibility of handling hazards
  - E.g., static branch prediction and code scheduling to reduce or prevent all hazards
  - Reorder instructions into issue packets
  - No dependencies with a packet

Example: Two-Issue RISC-V Processor

Static multiple issue example

- Two-issue packets per cycle
  - ➢One for ALU/branch inst., and then
  - ➢one for load/store inst.
  - ➢The instructions are paired and aligned to 64-bit boundary
  - ➢Pad an unused instruction with a nop inst.
- Simplify the design of inst. decoding and issue

- In some designs, compilers are responsible for removing *all* hazards in static multi-issue processors
  - ➢Schedule inst., and insert NOPs to avoid hazards
- In other designs, HW detects data hazards and stalls if hazards occur
  - ➢Compilers do their best to avoid all dependencies within an inst. packet

An issue packet

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

FIGURE 4.69 Static two-issue pipeline in operation. The ALU and data transfer instructions are issued at the same time. Here we have assumed the same five-stage structure as used for the single-issue pipeline. Although this is not strictly necessary, it does have some advantages. In particular, keeping the register writes at the end of the pipeline simplifies the handling of exceptions and the maintenance of a precise exception model, which become more difficult in multiple-issue processors

Example: Two-Issue RISC-V Processor (Datapath)

- Additional resources are needed for the datapath to support multiple issue (Fig. 4.70)

- Require extra ports in register file
  - ➢to issue an ALU and a data transfer inst. in parallel
  - ➢I.e., read two registers for ALU, and two more for a store (one write port for ALU and one for a load)
- Need a separate adder
  - ➢to calculate the effective addr. for data transfers

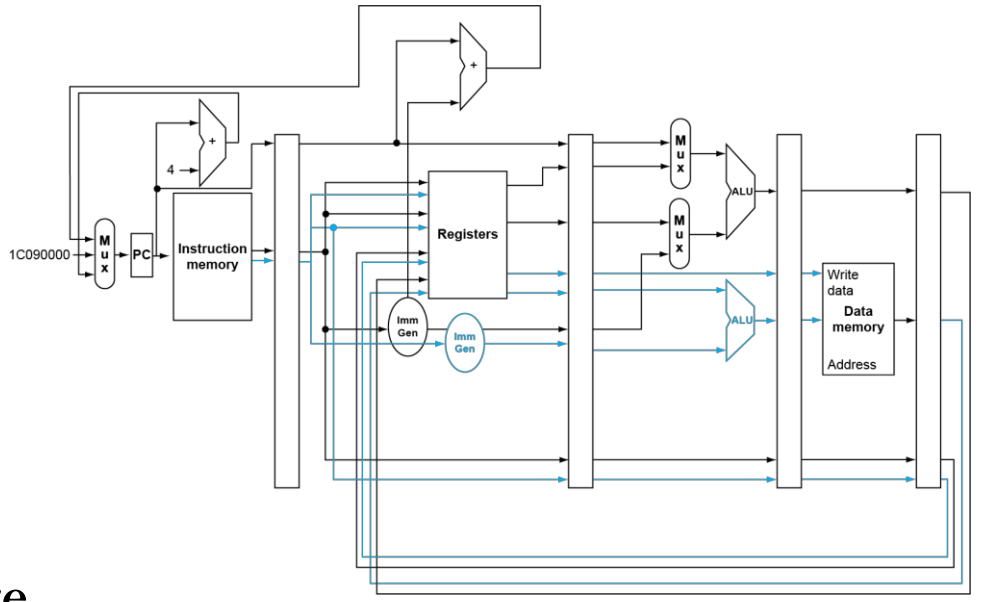


FIGURE 4.70 A static two-issue datapath. The additions needed for double issue are highlighted: another 32 bits from instruction memory, two more read ports and one more write port on the register file, and another ALU. Assume the bottom ALU handles address calculations for data transfers and the top ALU handles everything else

Example: Two-Issue RISC-V Processor (Hazards)

In the design of the previous, single-issue five-stage pipeline

- A load inst. has a use latency (1 bubble)
- ALU inst. has NO use latency (thanks to the forwarding)

In the dual-issue five-stage pipeline

- Load-use hazard
  - Still one cycle use latency (the next two inst. cannot use the load result without stalling)
- EX hazard
  - Cannot use ALU result in the paired load/store (in same packet)
  - add x10, x0, x1
ld x2, 0(x10)
  - Split into two packets, or effectively a stall
- Ambitious compiler/hardware scheduling techniques are needed to enjoy the advantage of multiple issues

An issue packet

N+4 Load is available at the 3rd cycle

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

Clock cycle 1

Clock cycle 2

Clock cycle 3

Example: Code Scheduling for Multiple Issue

- A loop code (top)
- The schedule of the code on the two-issue RISC-V pipeline (VLIW) (bottom)
  - Assume branches are predicted (ctrl hazards handled by HW)
- Dependencies exist
  - Among the first three inst.
  - Between the last two inst.
- Only one pair of instructions has both issue slots used
- Four clocks for five inst.
  - IPC = 5/4 = 1.25
  - c.f. peak IPC = 2
  - While counting IPC/CPI, NOPs are not considered

Less NOPs, better IPC

```
Loop:
    ld    x31,0(x20)      // x31=array element
    add   x31,x31,x21     // add scalar in x21
    sd    x31,0(x20)      // store result
    addi  x20,x20,-8      // decrement pointer
    blt   x22,x20,Loop    // branch if x22 < x20
```

Compiler schedules the code to avoid stalls

	ALU/branch	Load/store	cycle
Loop:	NOP	ld x31,0(x20)	1
	addi x20,x20,-8	NOP	2
	add x31,x31,x21	NOP	3
	blt x22,x20,Loop	sd x31,8(x20)	4

FIGURE 4.71 The scheduled code as it would look on a two-issue RISC-V pipeline. The empty slots are no-ops. Note that since we moved the addi before the sw, we had to adjust sw's offset by 4

Example: Code Scheduling for Multiple Issue (Loop Unrolling)

- Replicate (unroll) loop body to discover more inst. parallelism
  - Reduce loop-control overhead (the last two inst.: addi and blt)
  - The loop contains four copies each of lw, add, and sw, plus one addi and one blt

Compilers

- Use different registers per replication
  - Called "register renaming" to eliminate *not true* data dependencies
  - Dependency only exists among the inst. in the same iteration
- Avoid loop-carried "anti-dependencies"
  - Store followed by a load of the same register
  - Known as "name dependence," which is an ordering forced purely by the reuse of a name, rather than a real data dependence that is also called a true dependence
  - Reuse of a register name

- IPC = 14/8 = 1.75
  - Closer to 2, but at cost of registers and code size

Unrolled loop 4 times w/ reg. renaming:

```
ld   x28,0(x20)
add  x28,x28,x5
sd   x28,0(x20)

ld   x29,24(x20)
add  x29,x29,x21
sd   x29,0(x20)

ld   x30,16(x20)
add  x30,x30,x21
sd   x30,0(x20)

ld   x31,8(x20)
add  x31,x31,x21
sd   x31,0(x20)
addi x20,x20,-32
blt  x22,x20,Loop
```

	ALU/branch	Load/store	cycle
Loop:	addi x20,x20,-32	ld x28, 0(x20)	1
	nop	ld x29, 24(x20)	2
	add x28,x28,x21	ld x30, 16(x20)	3
	add x29,x29,x21	ld x31, 8(x20)	4
	add x30,x30,x21	sd x28, 32(x20)	5
	add x31,x31,x21	sd x29, 24(x20)	6
	nop	sd x30, 16(x20)	7
	blt x22,x20,Loop	sd x31, 8(x20)	8

FIGURE 4.72 The unrolled and scheduled code of Figure 4.71 as it would look on a static two-issue RISC-V pipeline. The empty slots are no-ops. Since the first instruction in the loop decrements x20 by 16, the addresses loaded are the original value of x20, then that address minus 4, minus 8, and minus 12

Dynamic Multiple Issue

- Dynamic multiple-issue processors are also known as *superscalar* processors

- In the simplest superscalar processor,
  - instructions are issued in order
  - It decides whether to issue 0, 1, 2 or more inst. in a given cycle

- For superscalar processors,
  - the code (whether scheduled or not) is guaranteed by the hardware to execute correctly
  - Avoiding structural and data hazards
  - Avoids the need for compiler scheduling
  - Though it may still help from compilers
  - Code semantics ensured by the CPU

Dynamic Pipeline Scheduling

- Dynamic pipeline scheduling
  - Hardware support for reordering the order of instruction *execution* to avoid stalls (opposite to the in-order issue)
- A dynamic pipeline scheduling processor
  - chooses which instructions to execute in a given clock cycle
  - while trying to avoid hazards and stalls
  - But, it commits (writes back) result to registers in order
    - A commit in which the results of pipelined execution are written to the programmer visible state in the same order that instructions are fetched

Why dynamic pipeline scheduling?
- Why not just let the compiler schedule code?
  - ✓ Not all stalls are predicable; e.g., cache misses
- Can't always schedule around branches
  - ✓ Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards

Example

```
ld   x31,20(x21)
add  x1, x31,x2
sub  x23,x23,x3
andi x5, x23,20
```

```
ld   x31,20(x21)
sub  x23,x23,x3
add  x1, x31,x2
andi x5, x23,20
```

- The dynamic pipeline scheduling processor can start the execution of sub (which does not depend on previous inst.)
  - while add is waiting for ld to improve performance
  - without waiting for the completion of lw and add inst.
  - Starts sub while add is waiting for ld (see above code)

Dynamic Scheduled Pipeline

- The pipeline is divided to 3 major units:

1. Fetch/Decode unit
2. Functional units
   - Each unit has buffers (reservation stations) to hold operands and operation
   - As soon as the operands are ready in buffers and FU is ready, the result is calculated
     - Done in the out-of-order fashion different from the fetched order
   - The result is sent to any reservation stations (RS) waiting for this particular result
   - as well as to the commit unit, which buffers the result
   - until it is safe to put the result into the register file or, for a store, into memory
3. Commit unit
   - The buffer in the commit unit is called reorder buffer (RB)
   - that is used to supply operands (similar to the forwarding logic in statically scheduled pipeline)
   - Once a result is committed to the register file, it can be fetched directly from there as normal pipeline

- Some other features (or units) are required
  - Memory accesses benefit from nonblocking caches, which continue servicing cache accesses during a cache miss; this is required for out-of-order execution processors

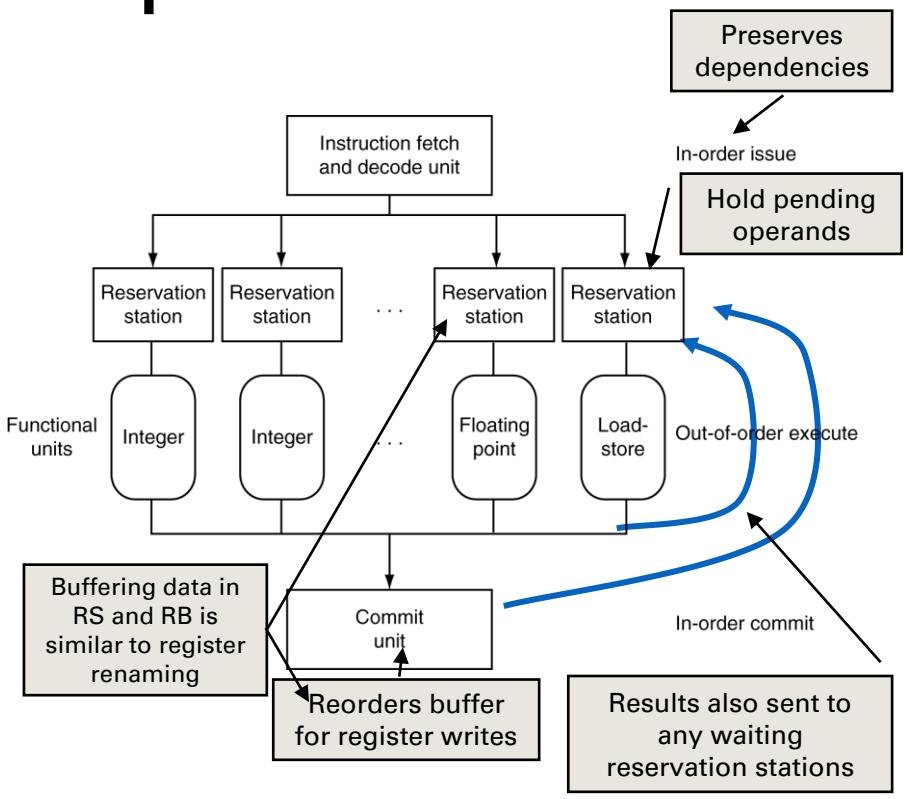


FIGURE 4.73 The three primary units of a dynamically scheduled pipeline. The final step of updating the state is also called retirement or graduation

<Design variants>
Fetch/Decode
In order or
Out of order
Exe (FU)
In order or
Out of order
Write back (Commit unit)
In order

Reservation station. A buffer within a functional unit that holds the operands and the operation
Commit unit. The unit in a dynamic or out-of-order execution pipeline that decides when it is safe to release the result of an operation to programmer visible registers and memory
Reorder buffer. The buffer that holds results in a dynamically scheduled processor until it is safe to store the results to memory or a register
Out-of-order execution. A situation in pipelined execution when an instruction blocked from executing does not cause the following instructions to wait

Power Efficiency

- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better → Current trend

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/ Speculation	Cores/ Chip	Power
Intel 486	1989	25 MHz	5	1	No	1	5 W
Intel Pentium	1993	66 MHz	5	2	No	1	10 W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29 W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75 W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103 W
Intel Core	2006	3000 MHz	14	4	Yes	2	75 W
Intel Core i7 Nehalem	2008	3600 MHz	14	4	Yes	2-4	87 W
Intel Core Westmere	2010	3730 MHz	14	4	Yes	6	130 W
Intel Core i7 Ivy Bridge	2012	3400 MHz	14	4	Yes	6	130 W
Intel Core Broadwell	2014	3700 MHz	14	4	Yes	10	140 W
Intel Core i9 Skylake	2016	3100 MHz	14	4	Yes	14	165 W
Intel Ice Lake	2018	4200 MHz	14	4	Yes	16	185 W

FIGURE 4.74 Record of Intel Microprocessors in terms of pipeline complexity, number of cores, and power. The Pentium 4 pipeline stages do not include the commit stages. If we included them, the Pentium 4 pipelines would be even deeper

ARM A53 vs. Intel i7 920

Processor	ARM A53	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	100 milliWatts (1 core @ 1 GHz)	130 Watts
Clock rate	1.5 GHz	2.66 GHz
Cores/Chip	4 (configurable)	4
Floating point?	Yes	Yes
Multiple issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline stages	8	14
Pipeline schedule	Static in-order	Dynamic out-of-order with speculation
Branch prediction	Hybrid	2-level
1st level caches/core	16-64 KiB I, 16-64 KiB D	32 KiB I, 32 KiB D
2nd level caches/core	128-2048 KiB	256 KiB (per core)
3rd level caches (shared)	(platform dependent)	2-8 MB

ARM Coretex-A53 Pipeline

- A53 is dual-issue, statically scheduled superscalar with dynamic issue detection
  - An in-order pipeline; instruction can initiate execution when the results are available and proceeding inst. have initiated
- Integer instructions (nonbranch)
  - there are eight stages: F1, F2, D1, D2, D3/ISS, EX1, EX2, and WB
- Branch decisions
  - are made in ALU pipe 0, resulting in a branch misprediction penalty of eight cycles
- In comparison with the i7, the A53 consumes approximately 1/200 the power for a quad-core processor

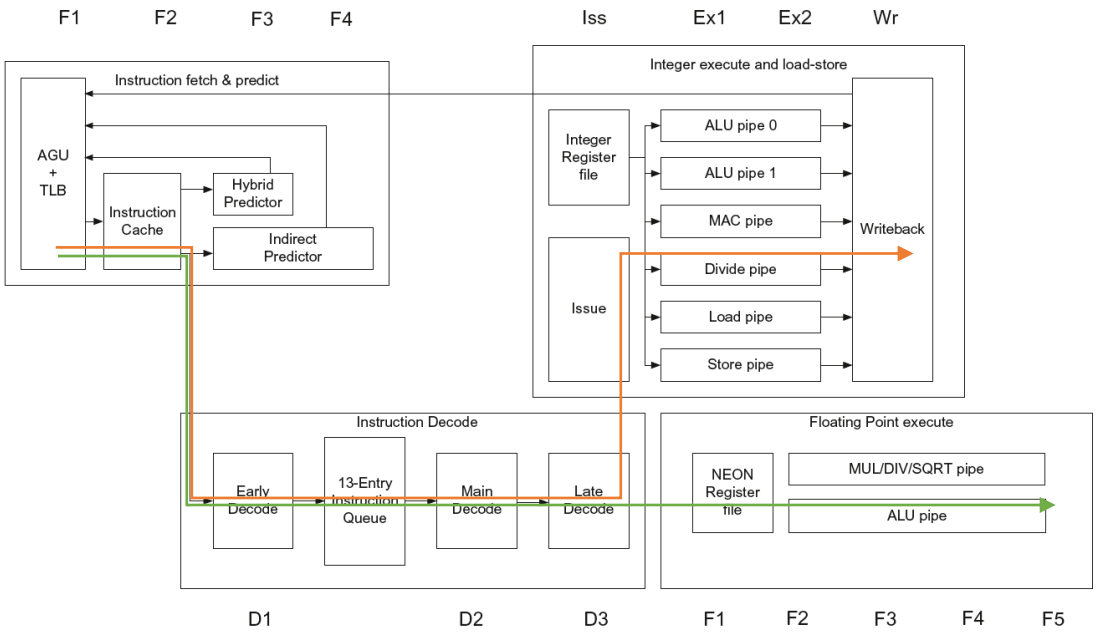


FIGURE 4.75 The basic structure of the A53 integer pipeline has eight stages: F1 and F2 fetch the instruction, D1 and D2 do the basic decoding, and D3 decodes more complex instructions and is overlapped with the first stage of the execution pipeline (ISS). After ISS, the Ex1, EX2, and WB stages complete the integer pipeline. Branches use four different predictors depending on type. The floating-point execution pipeline is 5 cycles deep in addition to the 5 cycles needed for fetch and decode, yielding 10 stages total. AGU stands for address generation unit and TLB for transaction lookaside buffer (See Chapter 5). The NEON unit performs the ARM SIMD instructions of the same name. (From Hennessy JL, Patterson DA: Computer architecture: A quantitative approach, 6e, Cambridge MA, 2018, Morgan Kaufmann.

More about the in-order pipeline. If the next two instructions are dependent, both can proceed to the appropriate execution pipeline, but they will be serialized when they get to the beginning of that pipeline. When the pipeline issue logic indicates that the result from the first instruction is available, the second instruction can issue

Fallacy and Pitfall

- Pipelining is easy (!)
  - The basic idea is easy
  - The devil is in the details
  - e.g., detecting data hazards and design of the controls

- Poor ISA design can make pipelining harder
  - E.g., complex instruction sets (VAX, IA-32)
  - Significant overhead to make pipelining work
  - Addressing modes that update registers complicate hazard detection
    - E.g., complex addressing modes

Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall

Questions?