(Parallel) Computing Challenges

- **Multi-core/processor**
  - Reduce communication/synchronization overload
  - Load balance
  - Cache coherence (different caches with same values)

- **Single Core**
  - Memory hierarchy
  - Instruction synchronization
    - **Arithmetics:** associativity (instruction ordering matters)
    - Advanced instructions: hardware and compilers

- **I/O:** redundant arrays of inexpensive disks (RAID: multiple resources for I/O - goes for fault tolerance as well)

The Difficulty of Parallel Processing Programs

- We must get **efficiency** - else use single processor
- Instruction level parallelism done by the compiler can help
  (out of order execution,etc)

Challenges

scheduling, load balancing, time for synchronization, overhead for communication between parts

Same for:

(1) 8 reporters writing together on a story
(2) 8 processors working in parallel

NOTE: The more the processors/reporters the harder these problems.

The Difficulty of Parallel Processing Programs:

Amdahl's Law

- How much can a problem be improved?
- e.g.: We want a speed up of 90x faster with 100 processors

$$\text{Exec time after improvement} = \frac{\text{Exec time affected by improvement}}{\text{Amount of improvement}} + \text{Exec time unaffected}$$

$$\text{Speed-up} = \frac{\text{Exec time before}}{(\text{Exec time before} - \text{Exec time affected}) + \frac{\text{Exec time affected}}{100}}$$

Processors

$$\text{Speed-up} = \frac{1}{(1 - \text{Fraction time affected}) + \frac{\text{Fraction time affected}}{100}}$$

Assumption of perfect load balancing

Amdahl's Law : Example 1

We want a speed up of 90x faster with 100 processors

$$90 = \frac{1}{(1 - \text{Fraction time affected}) + \frac{\text{Fraction time affected}}{100}}$$

$$\rightarrow \text{Fraction time affected} = \frac{89}{89.1} = 0.999$$

So to get a speed-up of 90 from 100 processors the sequential part can only be 0.1%

Amdahl's Law : Example 2 (Bigger Problem)

- Suppose you want to perform simultaneously **two sums**:
  - sum of 10 scalar variables
  - matrix sum of pairs of 2D arrays with dimension 10x10

➡ **What speed up you get with 10 vs 100 processors?**

➡ **Calculate the speed-ups assuming matrices grow to 100x100**

Amdahl's Law : Example 2 (Bigger Problem)

Answer

Assume time **t** for the performance of an addition.
Then there are (for 100 processors) 100 additions that scale and 10 that do not.

$$\text{Time for 1 processor} = 100t + 10t = 110t$$

Amdahl's Law : Example 2 (Bigger Problem)

$$\text{Time for 10 processors} =$$

$$\text{Time after improvement} = \frac{\text{Exec time affected}}{\text{Amount of improvement}} + \text{Exec time unaffected}$$

$$= \frac{100t}{10} + 10t = 20t$$

$$\boxed{\text{Speed-up}_{10} = \frac{110t}{20t} = 5.5 \text{ (out of 10)}}$$

$$\text{Time for 100 processors} = \frac{100t}{100} + 10t = 11t$$

$$\boxed{\text{Speed-up}_{100} = \frac{110t}{11t} = 10 \text{ (out of 100)}}$$

For 110 numbers we get 55% of the potential speedup with 10 processors but only 10% of 100 processors.

Amdahl's Law : Example 2 (Bigger Problem)

What happens when we increase the matrix order?

$$\text{Time for 1 processor} = 10t + 10000t = 10010t$$

10 processors

$$\text{Exec time after improvement} = \frac{10000t}{10} + 10t = 1010t$$

$$\text{Speed-up}_{10} = \frac{10010t}{1010t} = 9.9 \text{ (out of 10)}$$

100 processors

$$\text{Exec time after improvement} = \frac{10000t}{100} + 10t = 110t$$

$$\text{Speed-up}_{100} = \frac{10010t}{110t} = 91 \text{ (out of 100)}$$

For larger problem size we get 99% with 10 processors and 91% with 100 processors

SCALING

- **STRONG SCALING**:
  speed-up on multiprocessors **without** increase on problem size (**Amdahl's law** considers the strong scaling)

- **WEAK SCALING**:
  speed-up on multiprocessors, while increasing the size of the problem *proportionally* to the increase in the number of processors

Amdahl's Law : Example 3 - Load Balancing

- In the previous example, in order to achieve the speed-up of 91 **(for the larger problem)** with 100 processors, we assumed that the load was perfectly balanced.

Perfect balance: each of the 100 processors has 1% of the work to do

What if: 1 of the 100 processors load is higher than all the other 99 ?
Calculate for increased loads of 2% and 5%

Amdahl's Law : Example 3 - Load Balancing

- If one has 2% of the parallel load then it must do 2% x 10000 (larger problem) = 200 additions
- The other 99 will share 9800

Since they operate simultaneously:

$$\text{Exec time after improvement} = \max\left(\frac{9800t}{99}, \frac{200t}{1}\right) + 10t = 210t$$

The speed-up drops to: $\text{Speed-up}_{100} = \frac{10010t}{210t} = 48$

$$\text{Speed-up}_{100}^{10000} \,|^{\text{NB}} = 48\% \text{ (instead of 91\%)}$$

- If one processor has 5% of the load, then it must perform 5% x 10000 = 500 additions

Fallacies and Pitfalls - I

PITFALL: Expecting the improvement of one aspect of a computer to increase *overall* performance by an amount proportional to the size of an improvement

Example: A program runs in 100s on a computer, with multiply operators responsible for 80s of this time. How much do I have to improve the speed of multiplication if I want to run 5 times faster?

$$\text{Exec. time after the improvement} = \frac{\text{Exec. time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected} =$$

$$= \frac{80\text{s}}{n} + (100 - 80) = \frac{80}{n} + 20$$

Asking for the left side to be 20 requires $n = \infty$

Fallacies and Pitfalls - II

FALLACY: Computers at low utilization use little power

Servers at 10% of load can utilize up to 2/3 of peak power

Towards Hardware for energy aware computing

Fallacies and pitfalls -III

PITFALL: Using a subset of performance equation as a performance metric

Example: use as an alternative to the Time metric for performance the MIPS (Million Instructions Per Second) - **is it enough ?**

$$\text{MIPS} = \frac{\#\text{Instruction}}{\text{Execution time}} \times 10^{-6} = \frac{\#\text{Instructions}}{\frac{\#\text{Instructions} \times \text{CPI}}{\text{Clockrate}}} \times 10^{-6} = \frac{\text{Clock rate}}{\text{CPI}} \times 10^{-6}$$

Problem: instruction count may vary, MIPS varies between programs

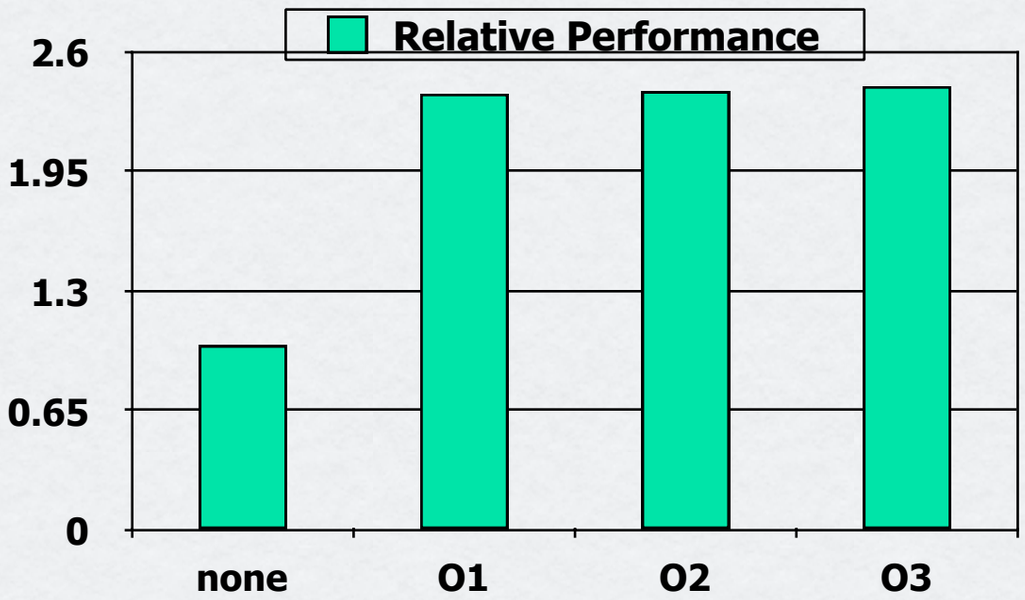
$\Longrightarrow$ no single MIPS rating for computers - hardware dependency

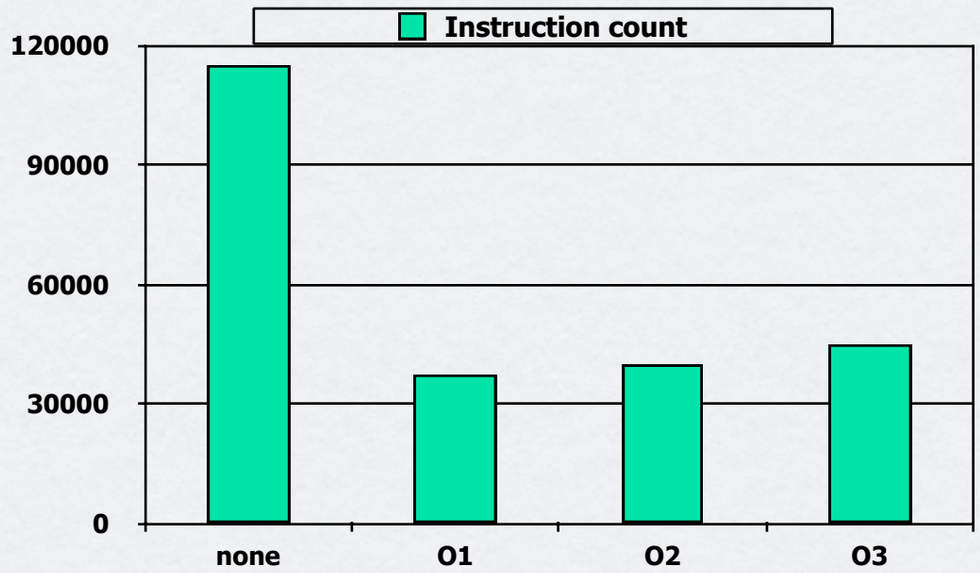
Execution time: the **only** valid/unimpeachable metric of performance

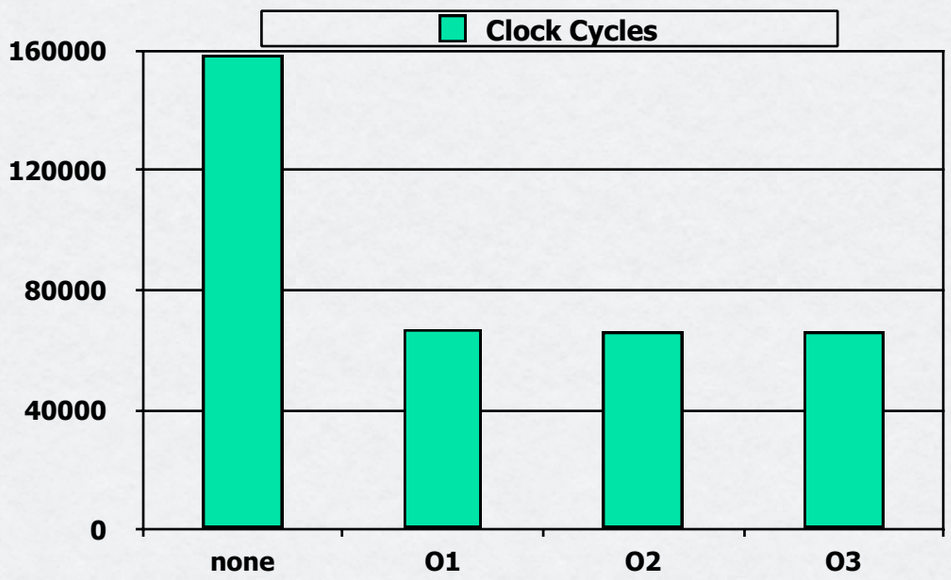
$$\text{Time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instructions}} \times \frac{\text{Seconds}}{\text{Clock cycles}}$$

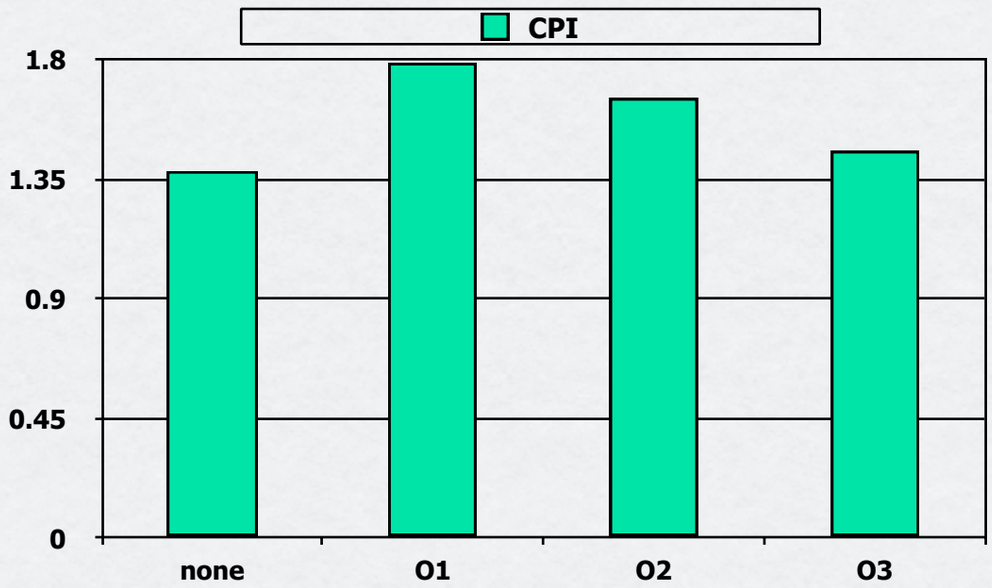
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

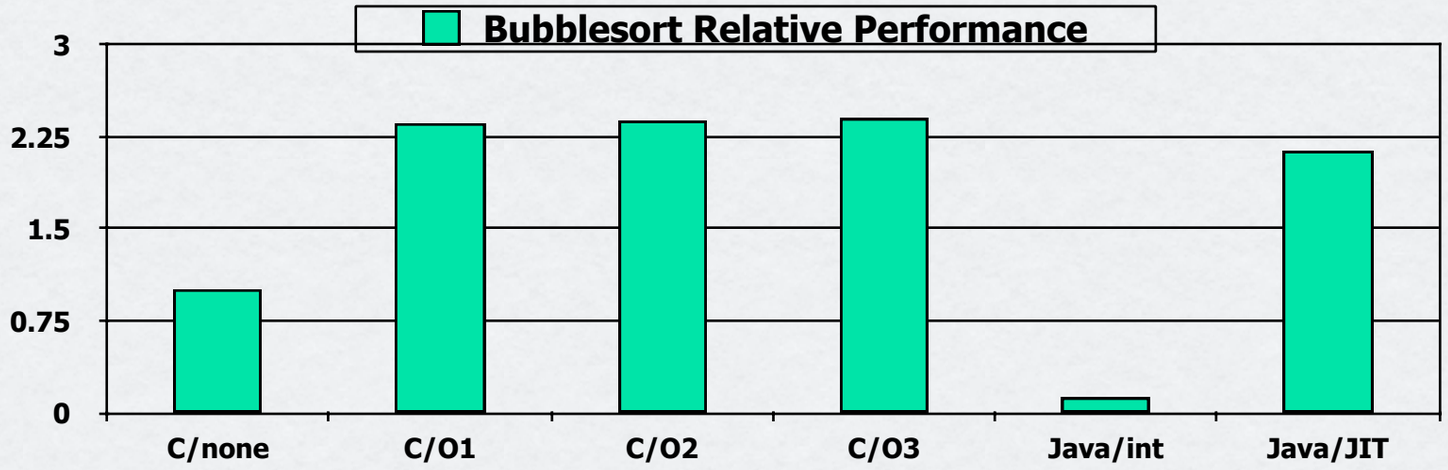


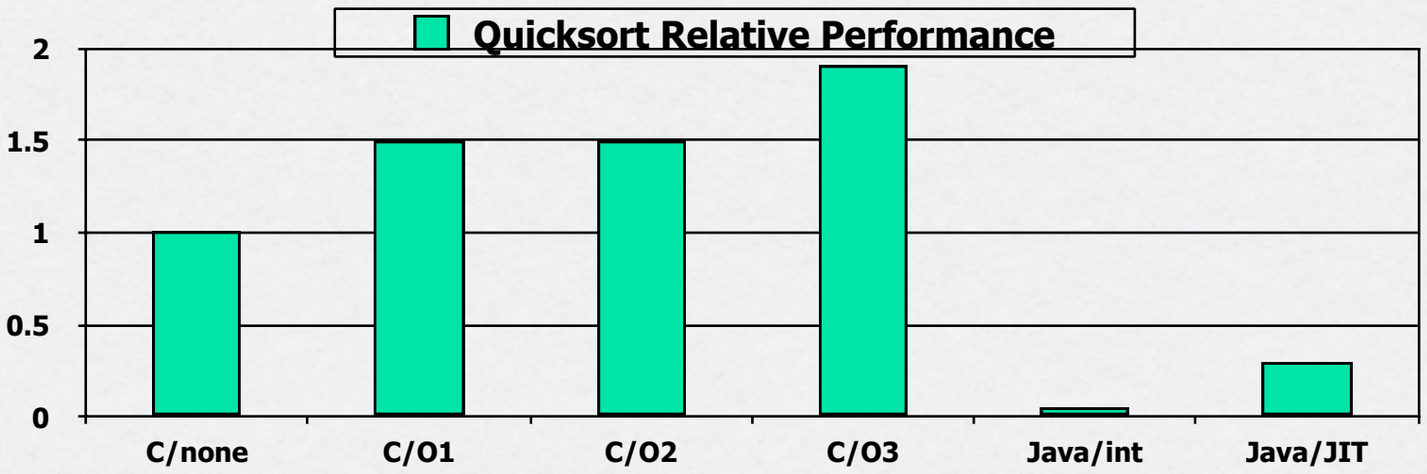


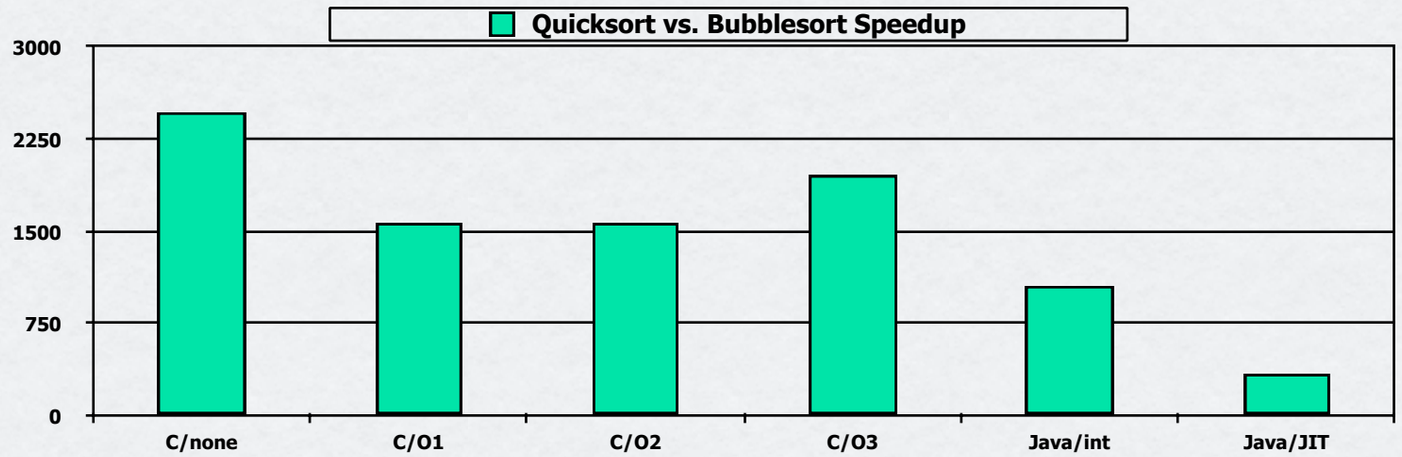




Effect of Language and Algorithm







Performance

Instruction count

Clock Cycle Time

Clock Cycles per Instruction (CPI)

Compiler + Instruction set architecture (ISA)

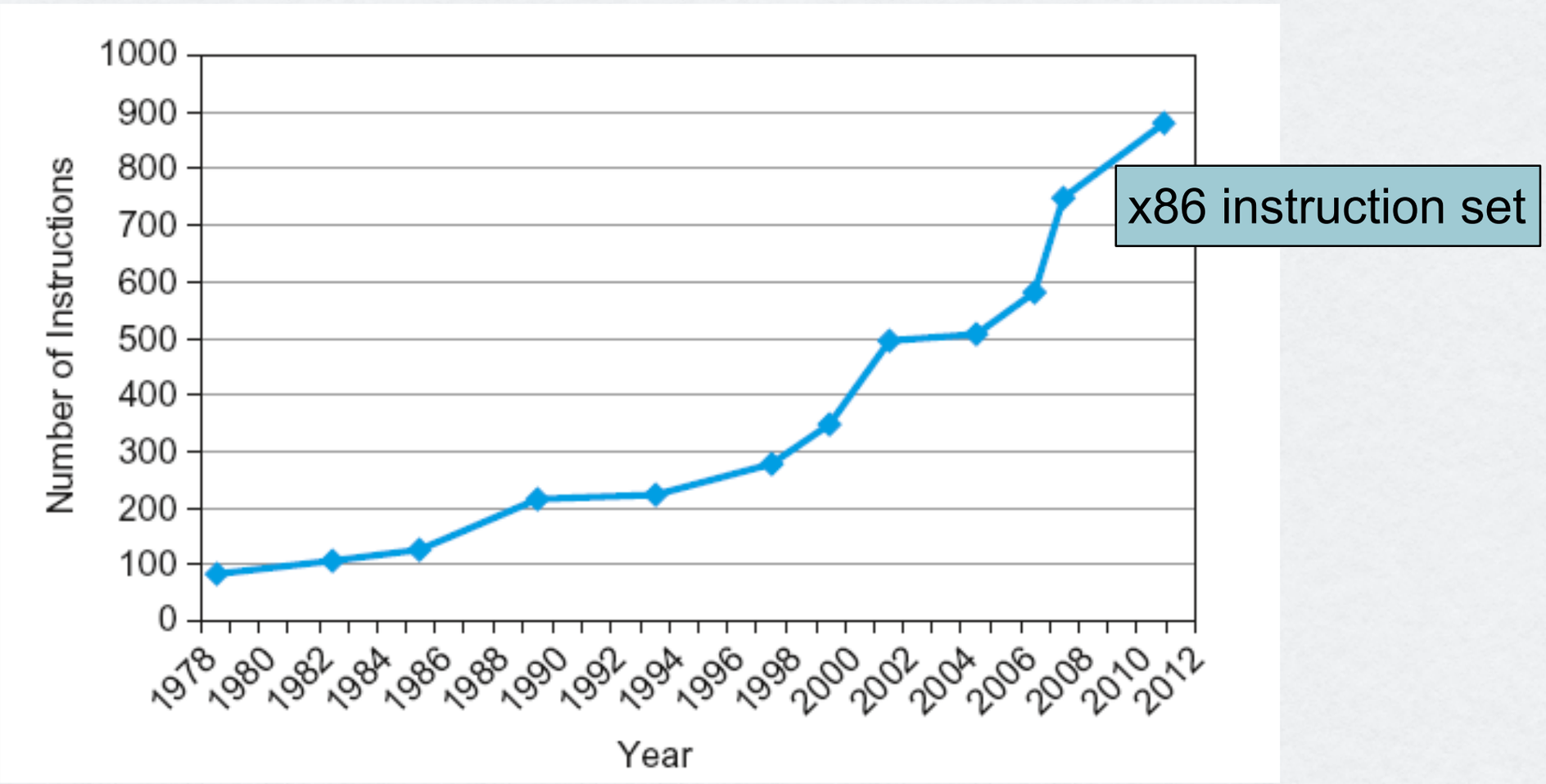
Processor Implementation

Fallacies

- **Powerful instruction ⇒ higher performance**
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
    - Compilers are good at making fast code from simple instructions
- **Use assembly code for high performance**
  - But modern compilers are better at dealing with modern processors
  - More lines of code ⇒ more errors and less productivity

Fallacies

- **Backward compatibility ⇒ instruction set doesn't change**
  - But they do accrete more instructions



Lessons Learnt

- **Instruction count and CPI are not good performance indicators in isolation**
- **Compiler optimizations are sensitive to the algorithm**
- Java/JIT compiled code is significantly faster than JVM interpreted - Comparable to optimized C in some cases
- **Nothing can fix a dumb algorithm!**

INSTRUCTIONS AND ARCHITECTURES

I. SOFTWARE

- Instruction: A word in a computer language

Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

ARCHITECTURE : MIPS

- **MIPS (M**icroprocessor without **I**nterlocked **P**ipeline **S**tages) architecture is a Reduced Instruction Set Computer (RISC) developed by MIPS Technologies

MIPS implementations are primarily used in embedded systems such as Windows CE devices, routers, residential gateways, and video game consoles such as the Sony Playstation, PlayStation 2 and PlayStation Portable. Until late 2006, they were also used in many of SGI's computer products. MIPS implementations were also used by Digital Equipment Corporation, NEC, Pyramid Technology, Siemens Nixdorf, Tandem Computers and others during the late 1980s and 1990s. In the mid to late 1990s, it was estimated that one in three RISC microprocessors produced was a MIPS implementation

The MIPS Instruction Set

- Used as the example throughout the notes
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- **Large share of embedded core market**
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs

A basic MIPS implementation

- Memory reference instructions: load word (**`lw`**) and store word (**`sw`**)
- Arithmetic-logical instructions: **`add`**, **`sub`**, **`and`**, **`or`** and **`slt`**
- Instruction branch: equal (**`beq`**) and jump (**`j`**) which we add last

- Two principles:
  - make the common case fast
  - simplicity favors regularity

- **Two key steps**
  - send program counter to memory – fetch instruction
  - read 1 or 2 registers, using fields of the instruction to select registers to read (for `lw` we need 1 register, for all others 2)

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
add a, b, c  # a gets b + c
```

- All arithmetic operations have this form

- ***Design Principle 1:*** Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic Example

- C code:

```
f = (g + h) - (i + j);
```

- Compiled MIPS code:

```
add t0, g, h   # temp t0 = g + h
add t1, i, j   # temp t1 = i + j
sub f, t0, t1  # f = t0 - t1
```

Representing Instructions

- **Instructions are encoded in binary**
  - Called machine code
- *MIPS instructions*
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!
- *Register numbers*
  - $t0 – $t7 are reg's 8 – 15
  - $t8 – $t9 are reg's 24 – 25
  - $s0 – $s7 are reg's 16 – 23

Instructions: language of the computer

- Operands are limited. Special locations build directly on hardware, called registers
  - In MIPS a register has 32 bits

Design Principle 2: Smaller is faster (only 32 registers)

- A very large number of registers may increase the clock cycle simply because it takes electronic signals longer to travel

More registers vs Fast clock cycle

- Effective use of registers is critical for computer performance

Register Operand Example

- C code:

f = (g + h) - (i + j);

  - f, …, j in $s0, …, $s4

- Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4
- MIPS is Big Endian
  - Most-significant byte at least address of a word
  - *c.f.* Little Endian: least-significant byte at least address

Memory operands

load: Data transfer instruction from memory to a register

format

name of operation

register to be loaded

constant offset

register to access memory

$g = h + A[8]$

```
lw    $t0, 8, ($s3)
add   $s1, $s2, $t0
```

- What is the rate of increase in #registers over time?
  - A1. Very fast: They increase as fast as Moore's law (doubling #transistors on a chip every 18 months)
  - A2. Very slow: Programs distributed in computer language. There is inertia in instruction set architecture and so registers increase only as fast as new sets become viable

Memory Operand Example 1

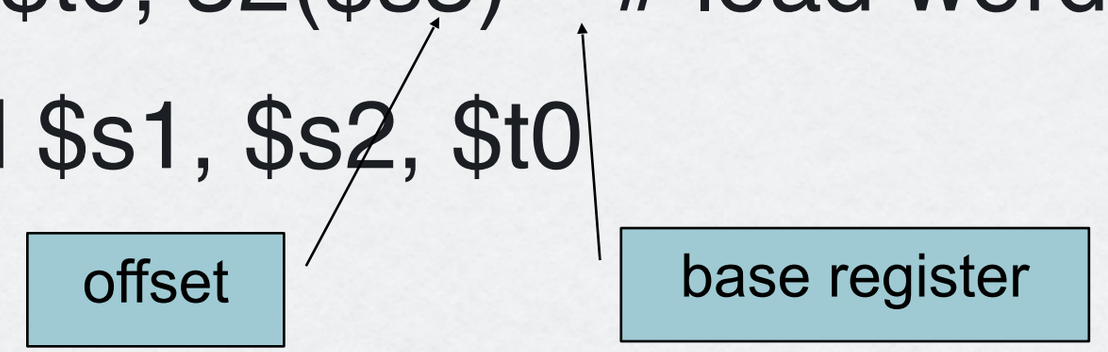
- C code:

```
g = h + A[8];
```

  - g in $s1, h in $s2, base address of A in $s3

- Compiled MIPS code:
  - Index 8 requires offset of 32
    - 4 bytes per word

```
lw  $t0, 32($s3)    # load word
add $s1, $s2, $t0
```



Memory Operand Example 2

- C code:

A[12] = h + A[8];

  - h in $s2, base address of A in $s3

- Compiled MIPS code:
  - Index 8 requires offset of 32

```
lw  $t0, 32($s3)    # load word
add $t0, $s2, $t0
```

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

Immediate Operands

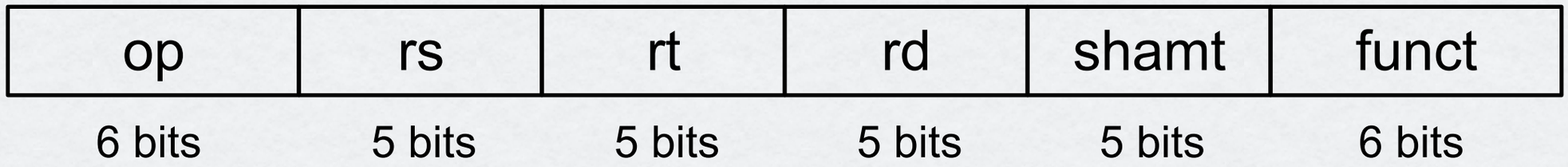
- Constant data specified in an instruction

  addi $s3, $s3, 4

- No subtract immediate instruction
  - Just use a negative constant

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

MIPS: R-type Instructions



- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)

R-type Instruction Example

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

add $t0, $s1, $s2

first and last fields tell the MIPS computer to perform addition

special	$s1	$s2	$t0	0	add
	number of register for first source operand	number of register for 2nd source operand	number of register to receive sum	unused in this instruction	

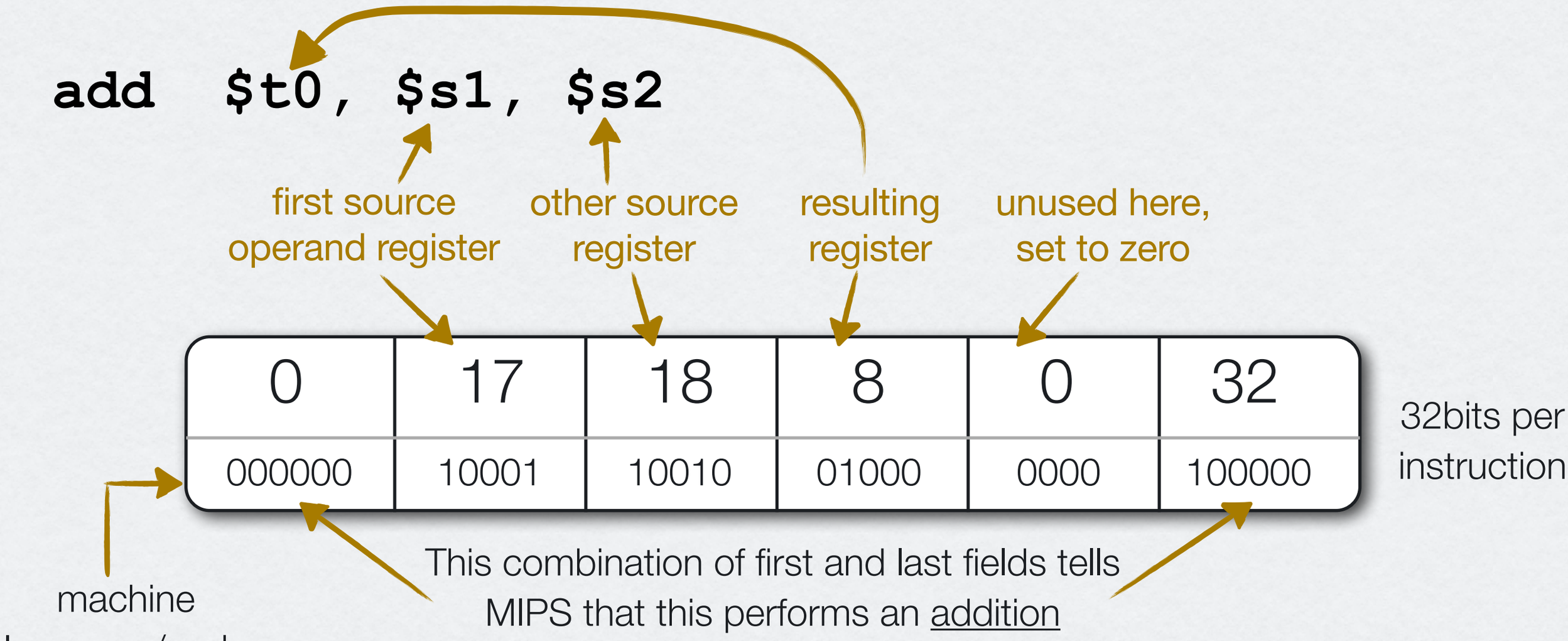
0	17	18	8	0	32

000000	10001	10010	01000	00000	100000

$$00000010001100100100000000100000_2 = 02324020_{16}$$

MIPS : Recap

- Big picture: Today's computers are built on two key principles:
  1. Instructions are represented as numbers
  2. programs are stored in memory to be read or written, just like numbers



Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

NOTE: C and Java use the notation 0xnnnn for hexadecimal numbers

MIPS I-type Instructions

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: offset added to base address in rs

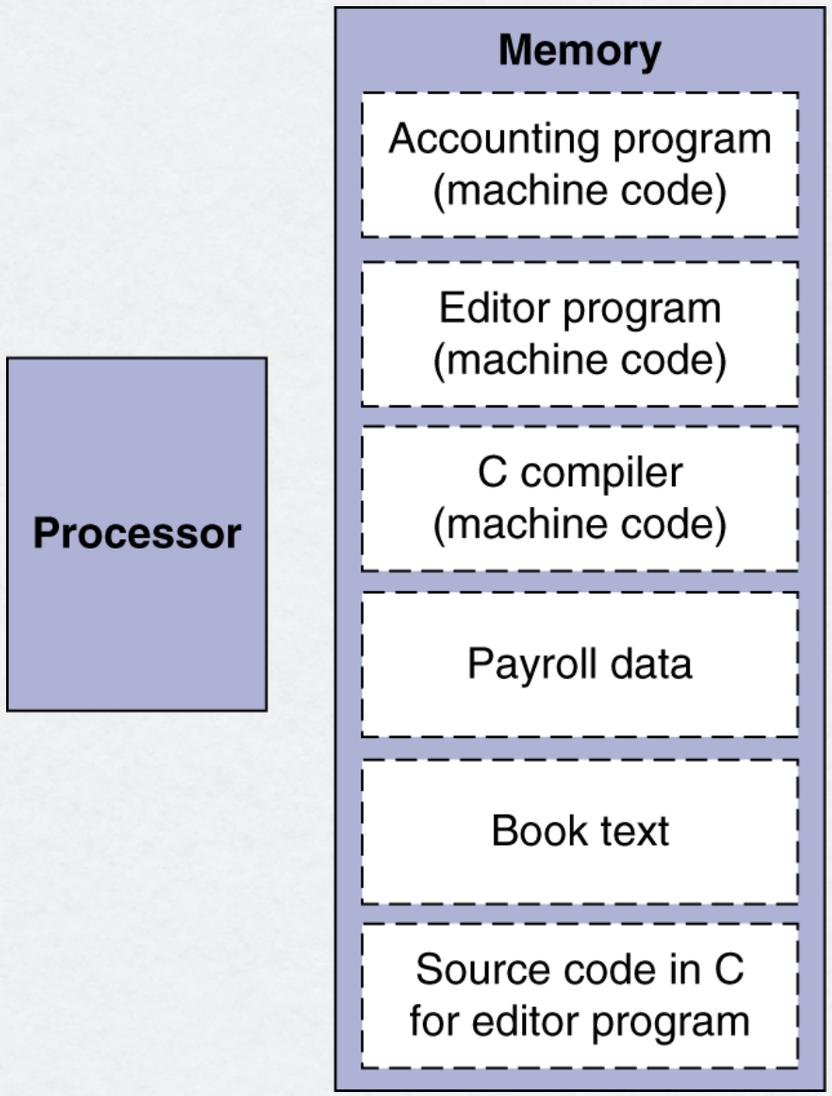
- ***Design Principle 4:*** Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

Conditional Operations ()

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- beq rs, rt, L1
  - if (rs == rt) branch to instruction labeled L1;
- bne rs, rt, L1
  - if (rs != rt) branch to instruction labeled L1;
- j L1
  - unconditional jump to instruction labeled L1

Stored Program Computers

The BIG Picture



- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

Parallelism and Instructions: Synchronization

- To know when a task has finished writing so that it is safe for another to read, the tasks need to synchronize

- **No synchronization:** danger of a data race, where the results of the program depend on how program was run
  - Consider the example of 8 reporters writing a common story

- In computers, lock and unlock are synchronization operations. Lock and unlock create mutual exclusions
  - Hardware primitives that atomically read and modify a memory location. Nothing can interpose between the read and write of the memory location.
  - Primitives are used by the system programmers to build synchronization libraries

Parallelism and Instructions: Synchronization

- A typical operation for building synchronization operations is the atomic exchange or atomic swap. This call **interchanges a value in a register for a value in memory**

- Example - Simple lock:

0 indicates that lock is free, 1 that it is unavailable

  - A processor tries to set the lock by doing an exchange of 1, which is in a register, with the memory address corresponding to the lock.
  - Return is:
    - 1 if some other processor had claimed access
    - 0 otherwise ⇒ value changed to 1

      (prevents anybody else from getting 0)

INSTRUCTIONS AND ARCHITECTURES

II. THE PROCESSOR

MIPS: The Chip

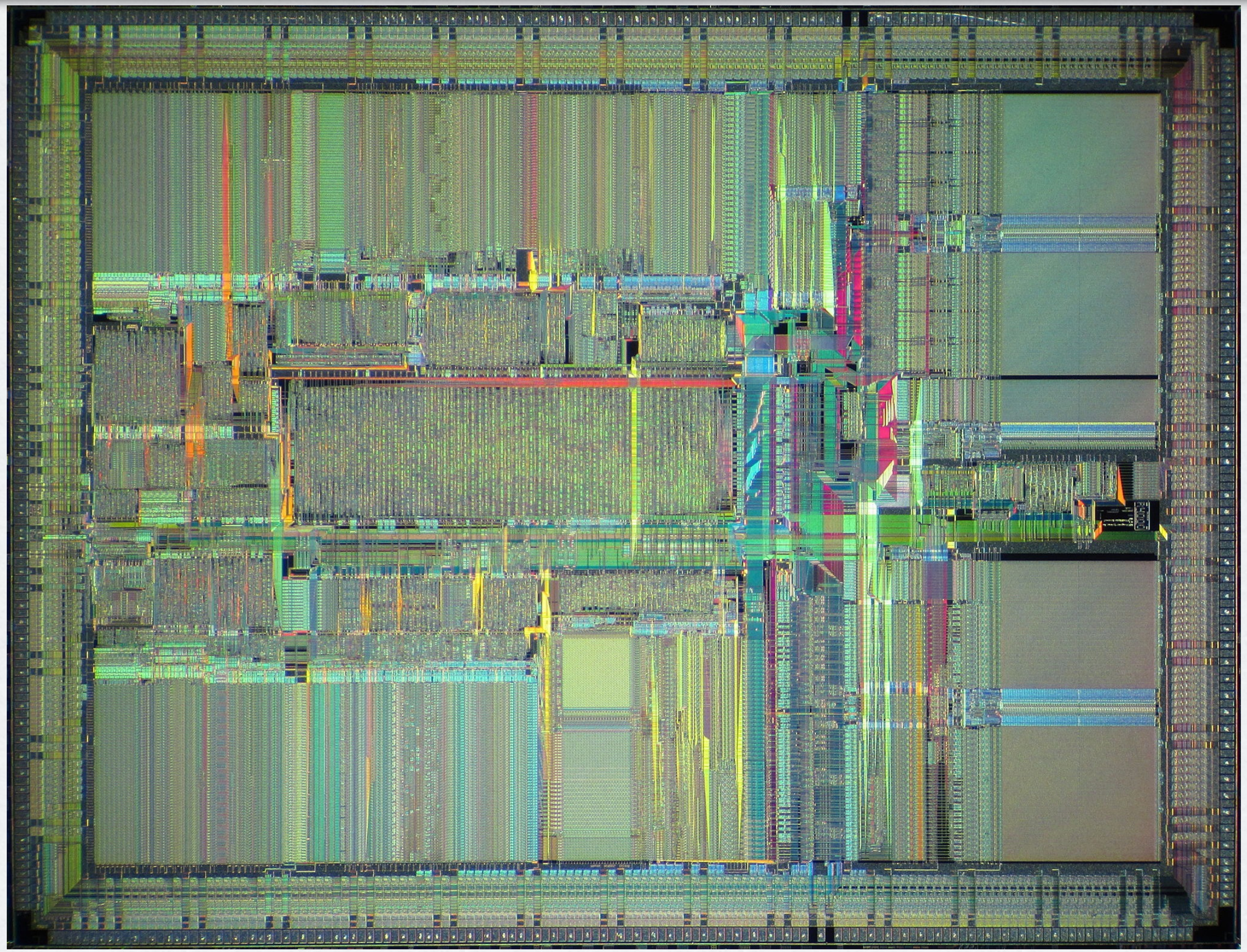
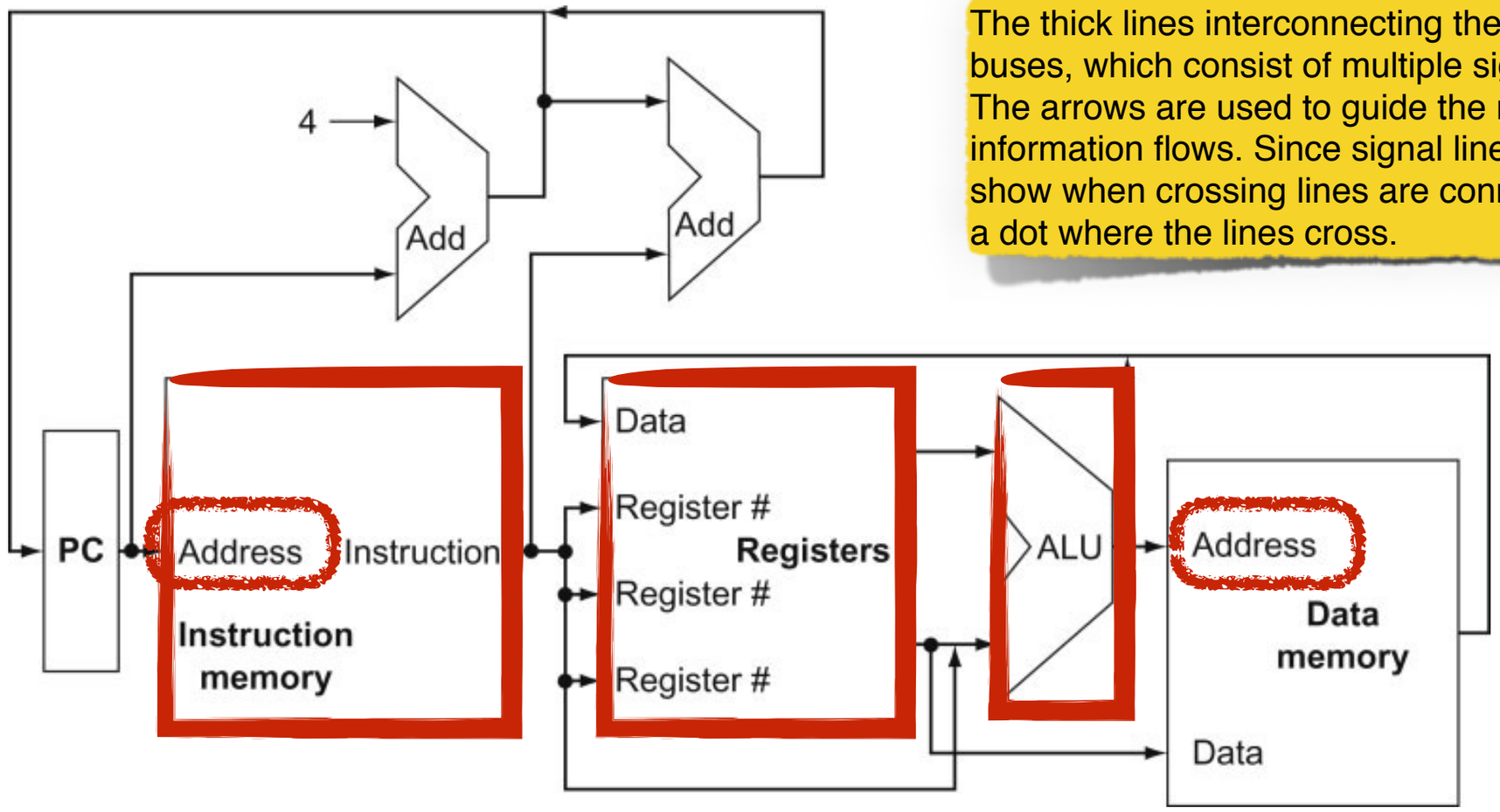


FIGURE 4.1 An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them.



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.

All **instructions start** by using the program counter to supply the **instruction address** to the instruction memory.

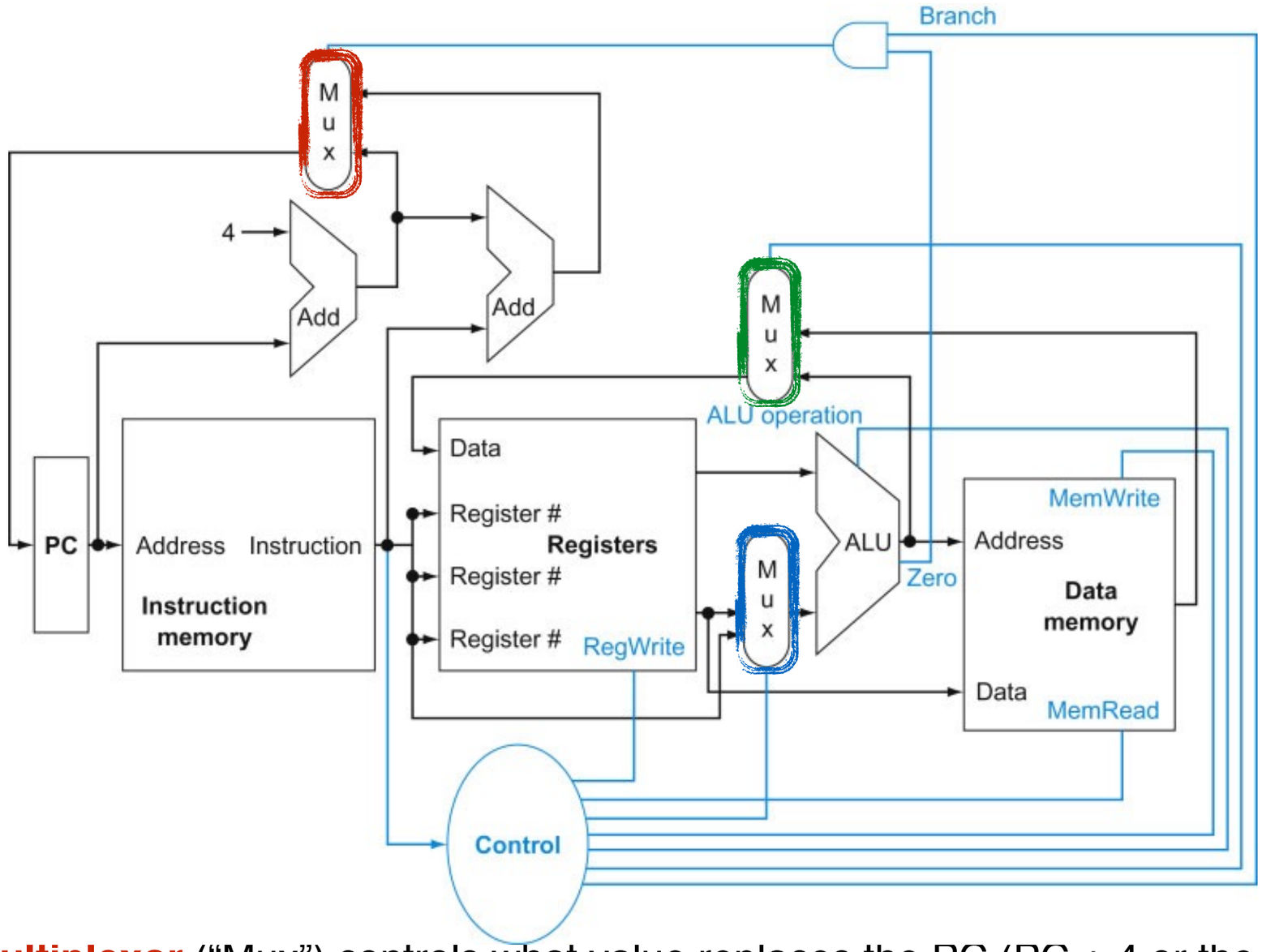
After the **instruction is fetched**, the register operands used by an instruction are specified by fields of that instruction.

Once the **register operands** have been fetched, they can be **operated on** to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or a compare (for a branch).

- ➡ If the instruction is an arithmetic-logical instruction, the result from the **ALU** must be written to a register.
- ➡ If the operation is a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers. 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 ALU (where the PC and branch offset are summed) or from an adder that increments the current PC by 4.

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



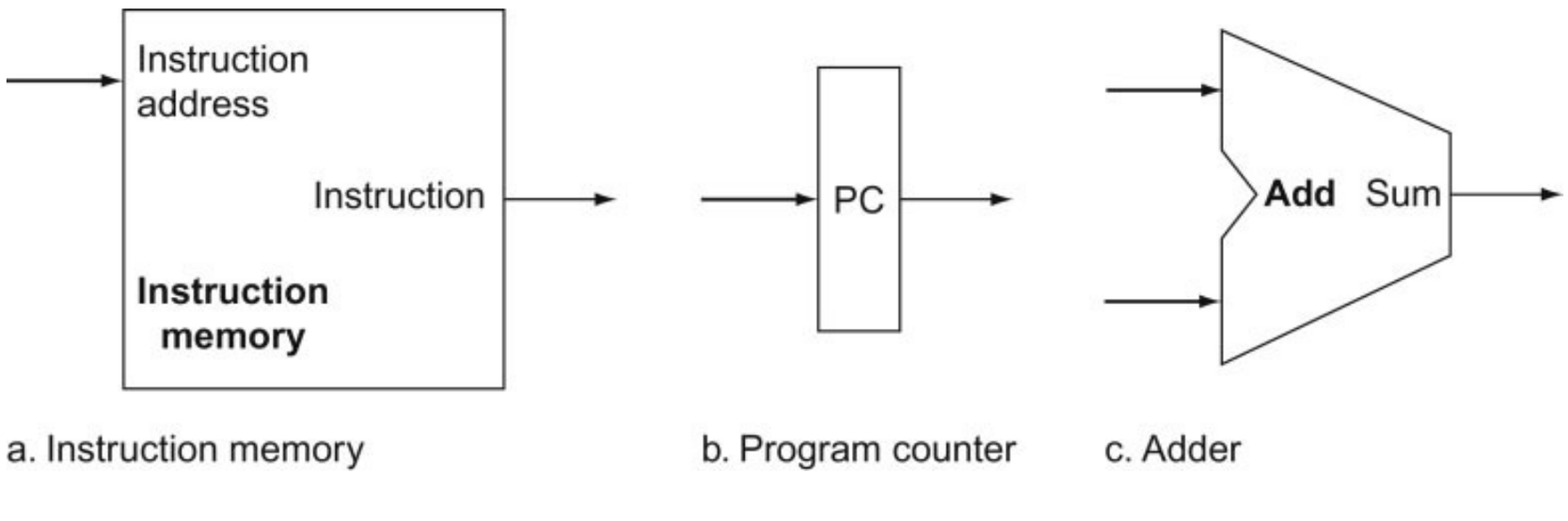
The added control lines are straightforward and determine the operation performed at the ALU, whether the data memory should read or write, and whether the registers should perform a write operation. The control lines are shown in color to make them easier to see.

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.

The **middle multiplexor,** whose output returns to the register file, is used to steer the output of the ALU (in the case of an arithmetic-logical instruction) or the output of the data memory (in the case of a load) for writing into the register file.

Finally, the **bottommost multiplexor** is used to determine whether the second ALU input is from the registers (for an arithmetic-logical instruction or a branch) or from the offset field of the instruction (for a load or store).

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.



a. Instruction memory b. Program counter c. Adder

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.

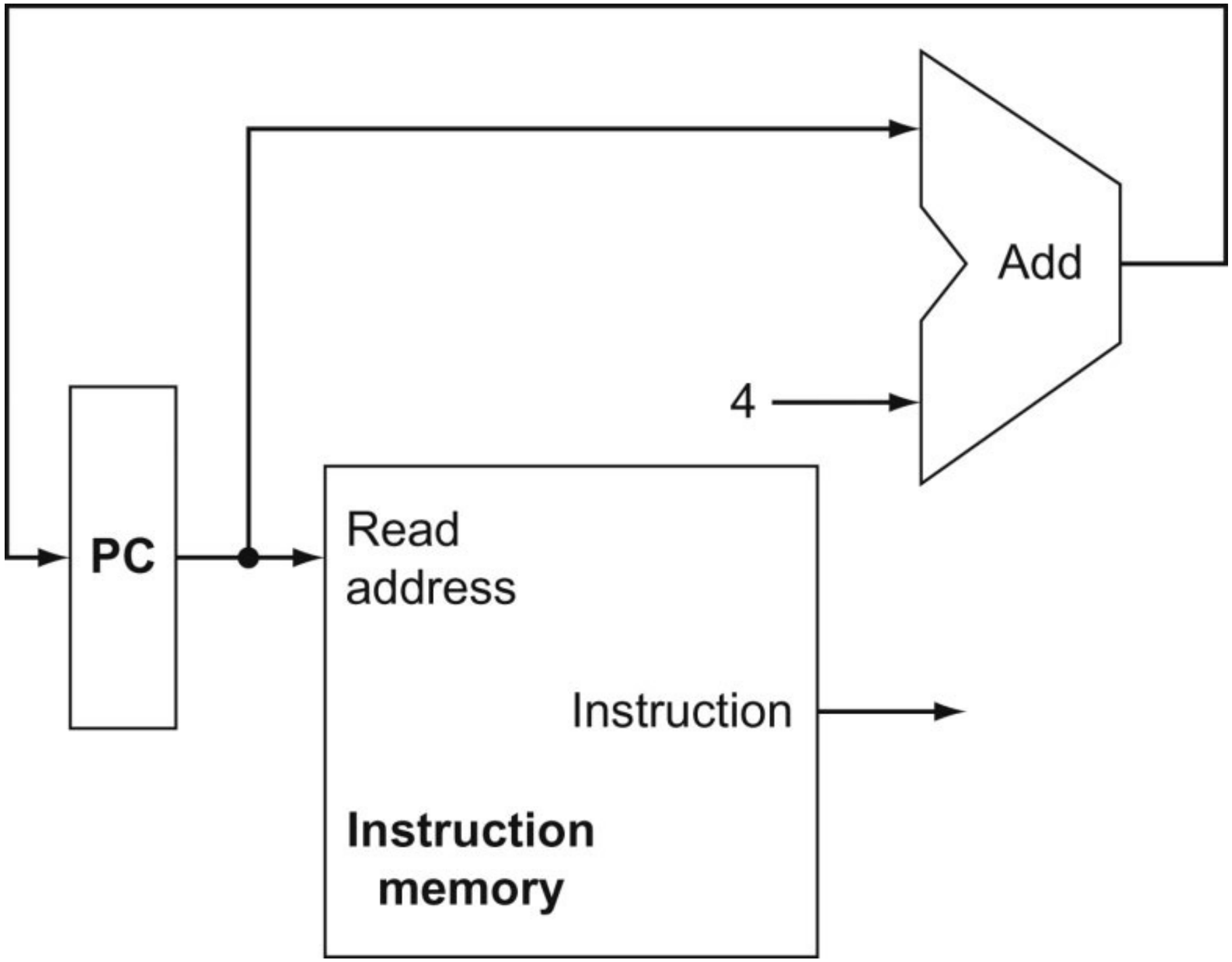
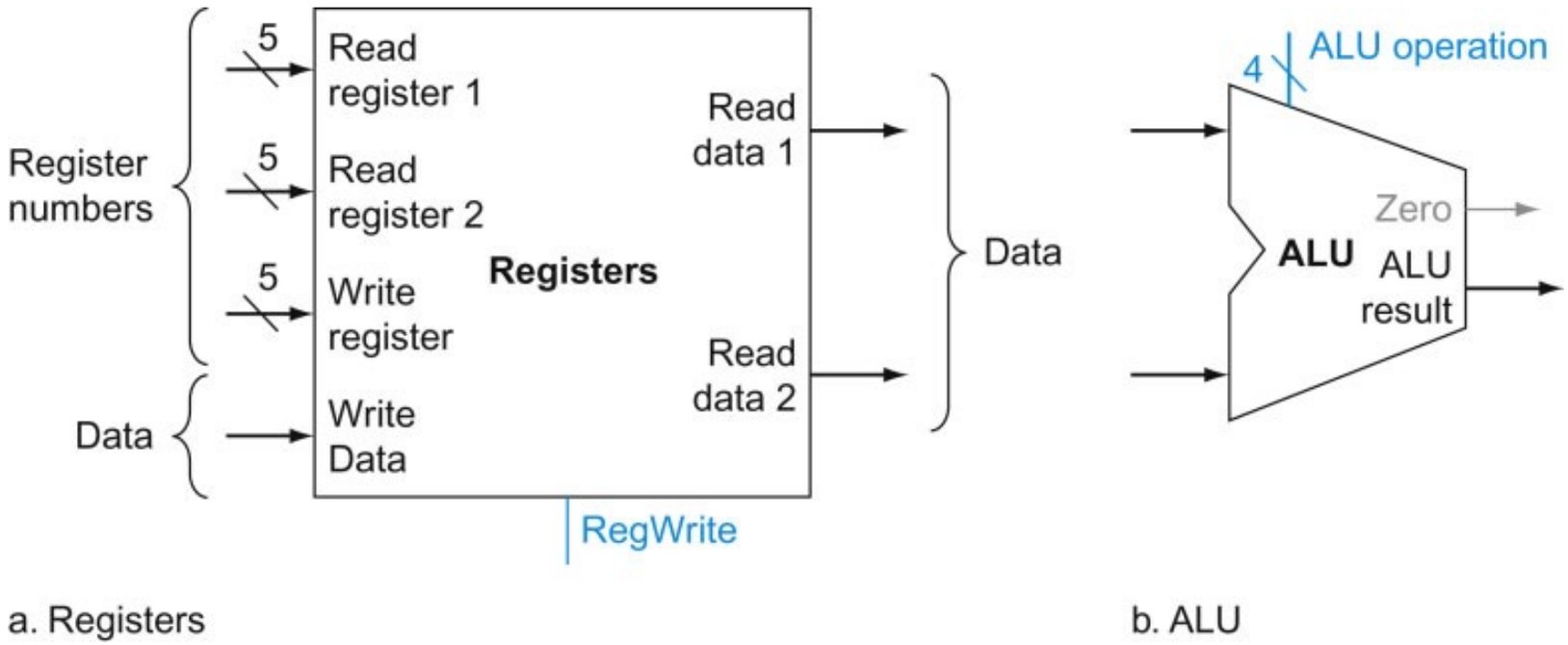


FIGURE 4.6 A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath.

FIGURE 4.7 The two elements needed to implement R-format ALU operations are the register file and the ALU.

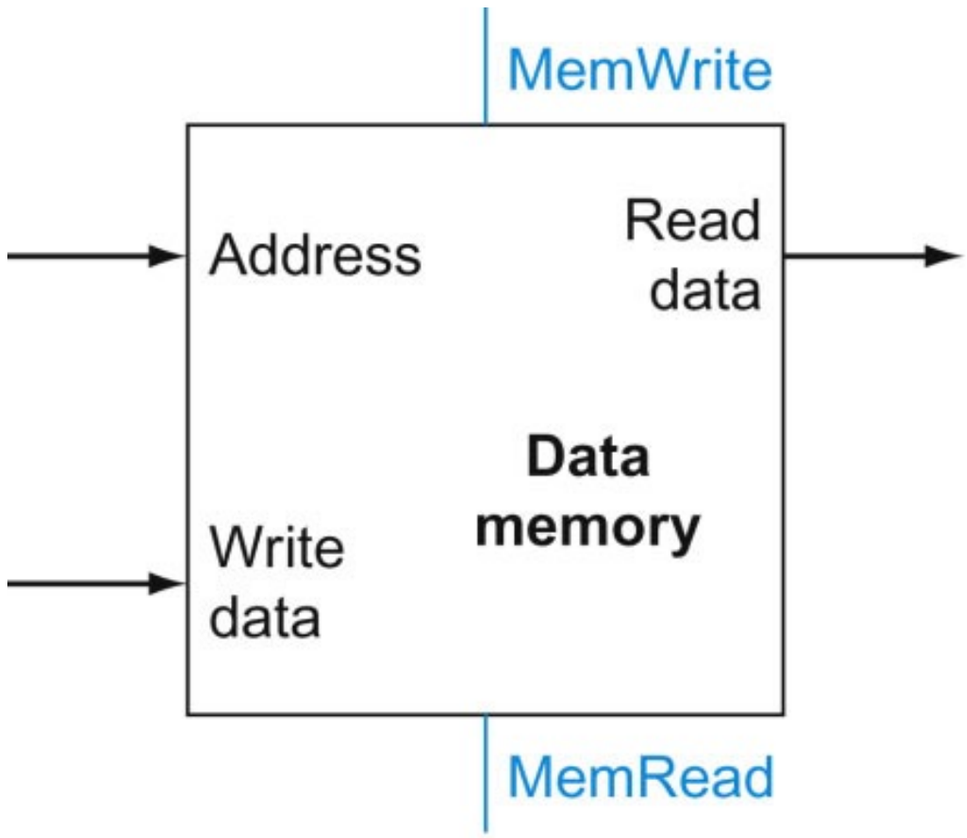


The **register file** contains all the registers and has two read ports and one write port. The design of multiported register files is discussed in Section B.8 of **Appendix B**. The register file always outputs the contents of the registers corresponding to the Read register inputs on the outputs; no other control inputs are needed. In contrast, a register write must be explicitly indicated by asserting the write control signal.

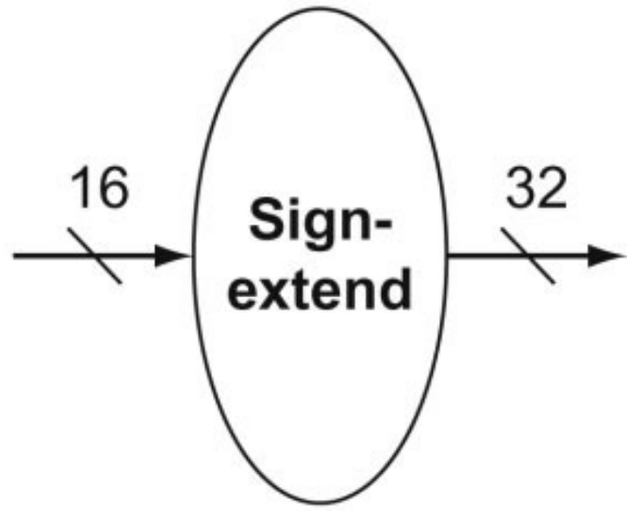
Remember that writes are edge-triggered, so that all the write inputs (i.e., the value to be written, the register number, and the write control signal) must be valid at the clock edge. Since writes to the register file are edge-triggered, our design can legally read and write the same register within a clock cycle: the read will get the value written in an earlier clock cycle, while the value written will be available to a read in a subsequent clock cycle. 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 B**. We will use the Zero detection output of the ALU shortly to implement branches. The overflow output will not be needed until Section 4.9, when we discuss exceptions; we omit it until then.

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 sign extension unit.



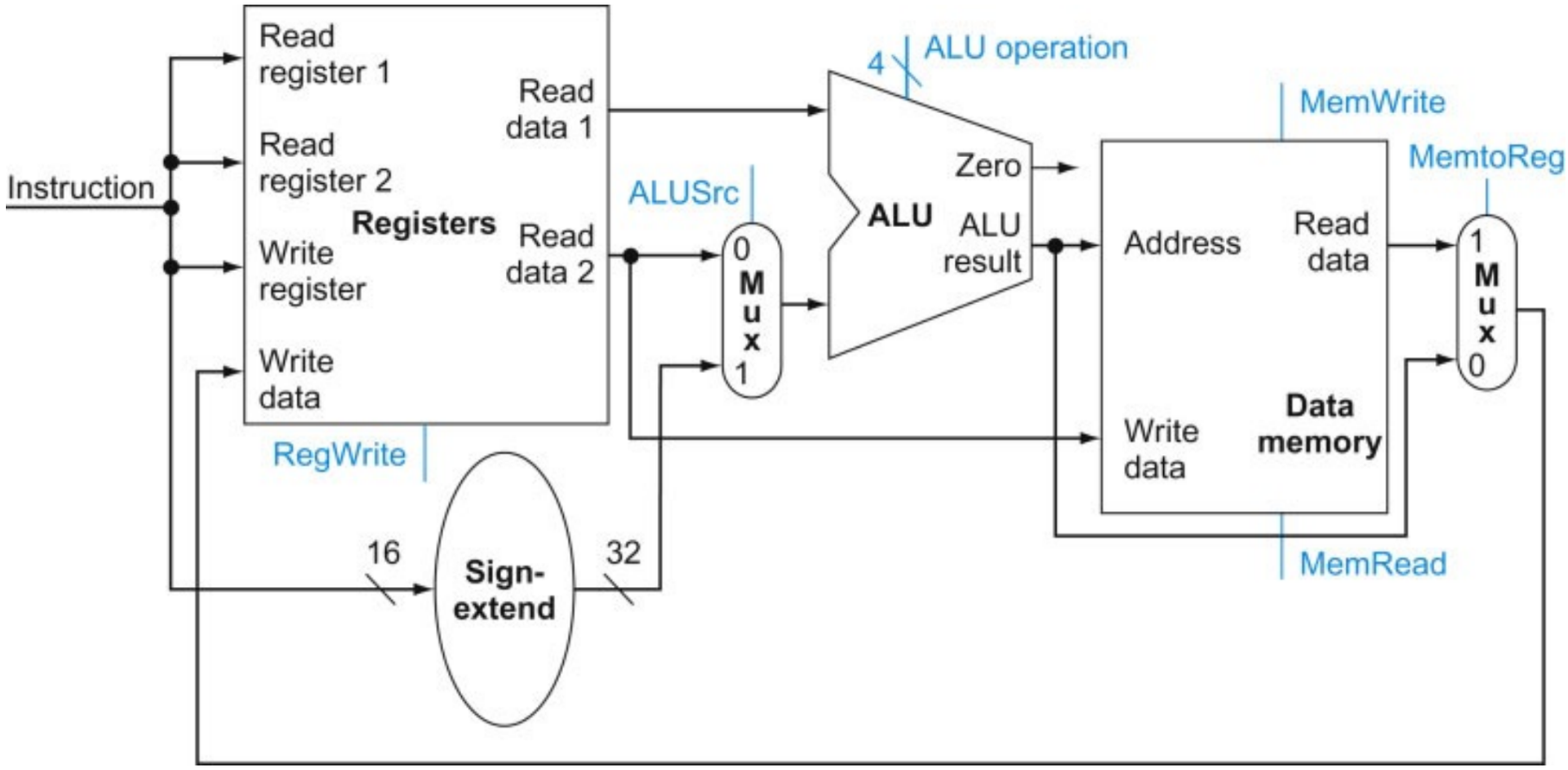
a. Data memory unit



b. Sign extension unit

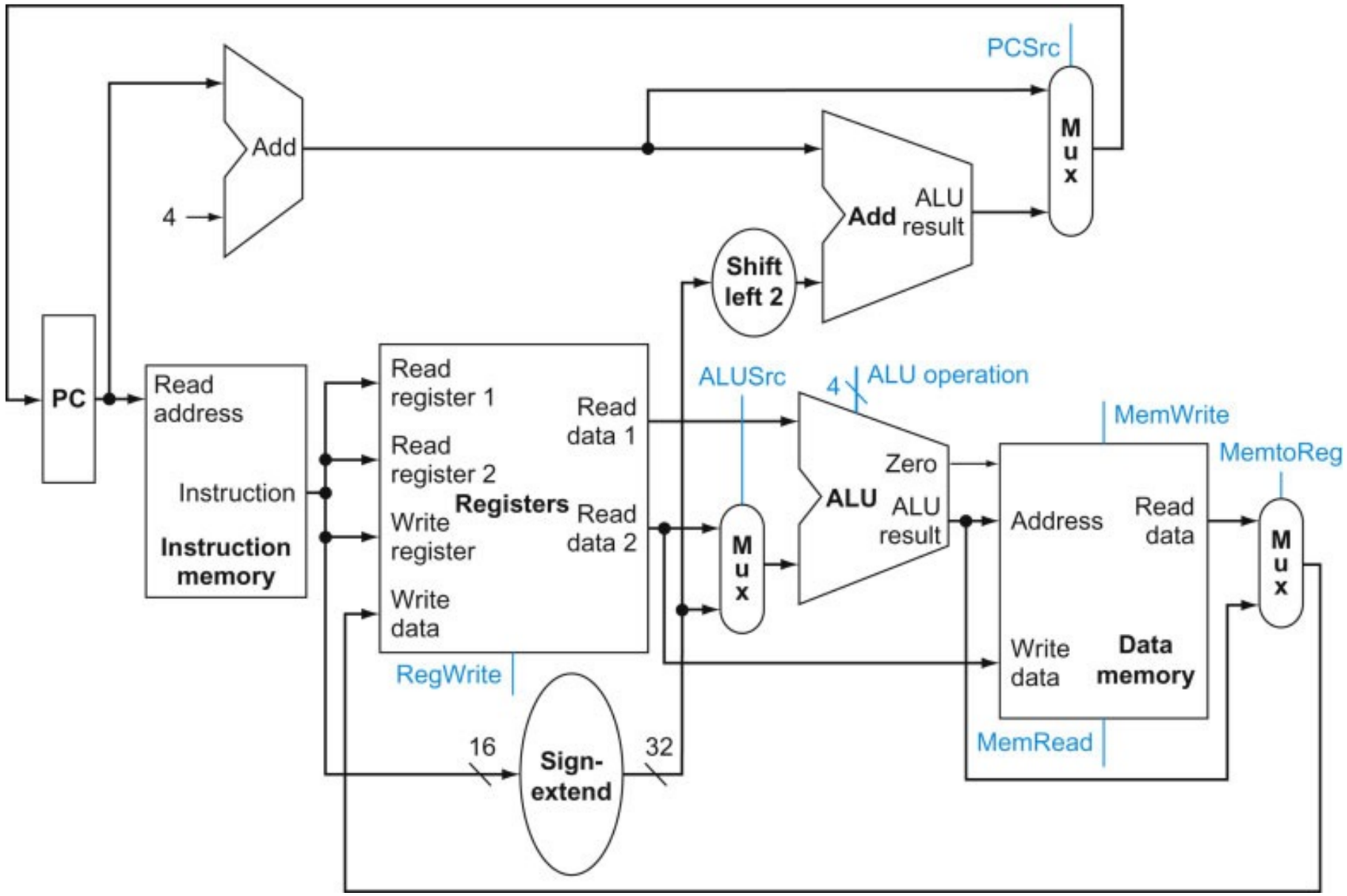
The memory unit is a state element with inputs for the address and the write data, and a single output for the read result. There are separate read and write controls, although only one of these may be asserted on any given clock. The memory unit needs a read signal, since, unlike the register file, reading the value of an invalid address can cause problems, as we will see in Chapter 5. The sign extension unit has a 16-bit input 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 B.8 of **Appendix B** for further discussion of how real memory chips work.

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.

FIGURE 4.11 The simple datapath for the core MIPS 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 word, ALU operations, and branches) in a single clock cycle.

Just one additional multiplexor is needed to integrate branches. The support for jumps will be added later.

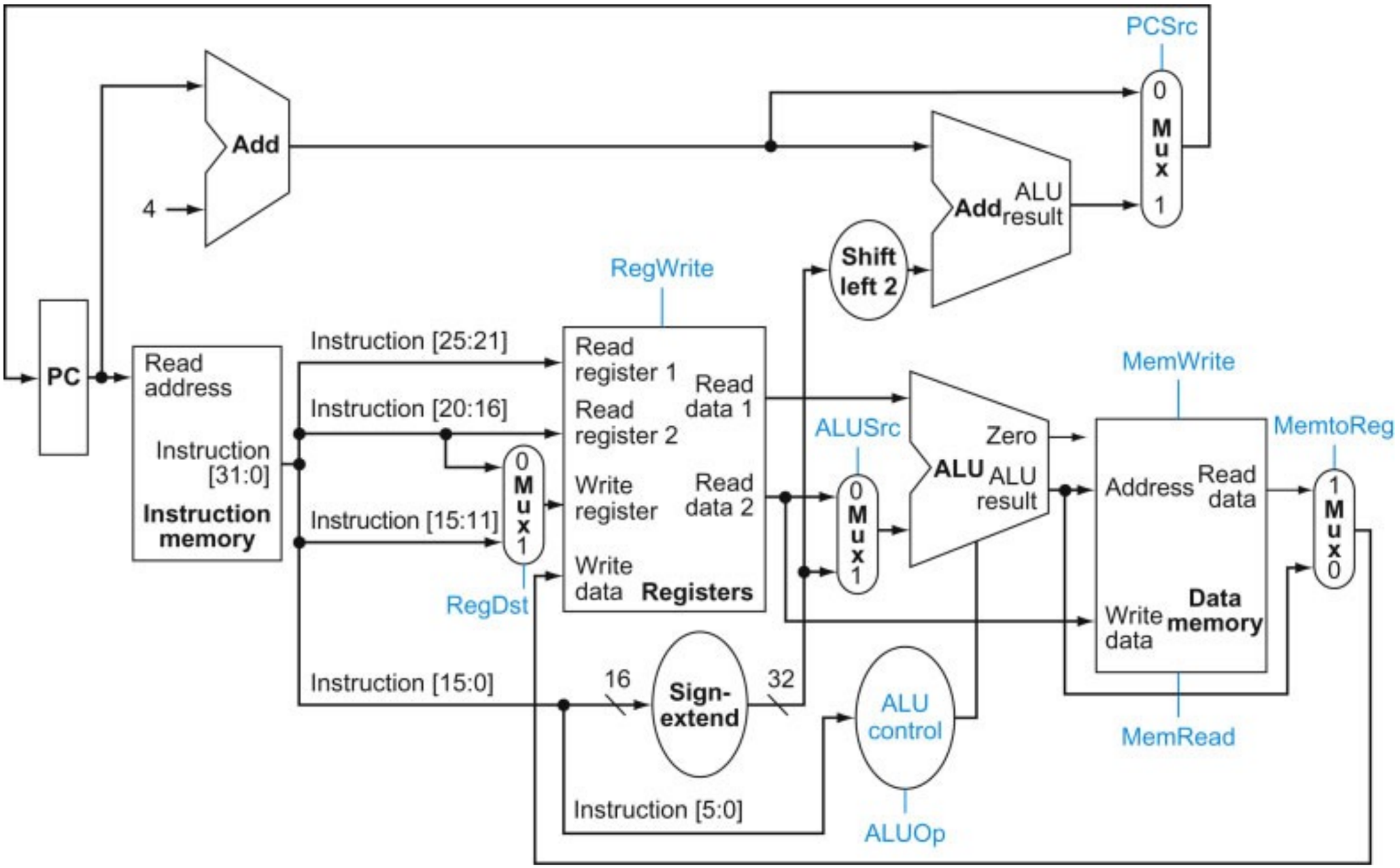
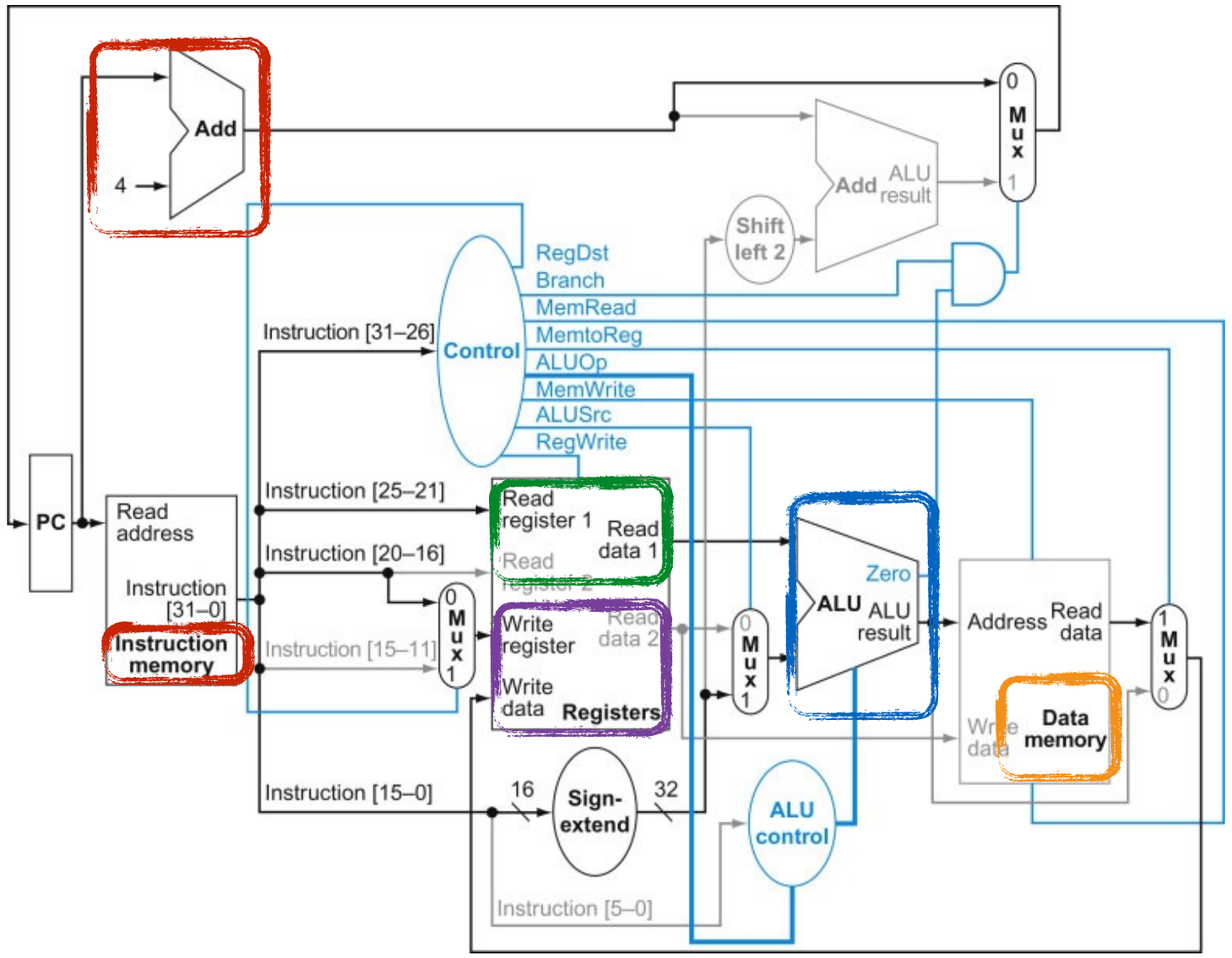


FIGURE 4.15 The datapath of Figure 4.11 with all necessary multiplexors and all control lines identified. The control lines are shown in color. The ALU control block has also been added. The PC does not require a write control, since it is written once at the end of every clock cycle; the branch control logic determines whether it is written with the incremented PC or the branch target address.

FIGURE 4.20 The datapath in operation for a load instruction (A store instruction would operate very similarly.)

```
lw  $t1, offset($t2)
```

The **control lines(blue)**, **datapath units, and connections (black)** that are active are highlighted.

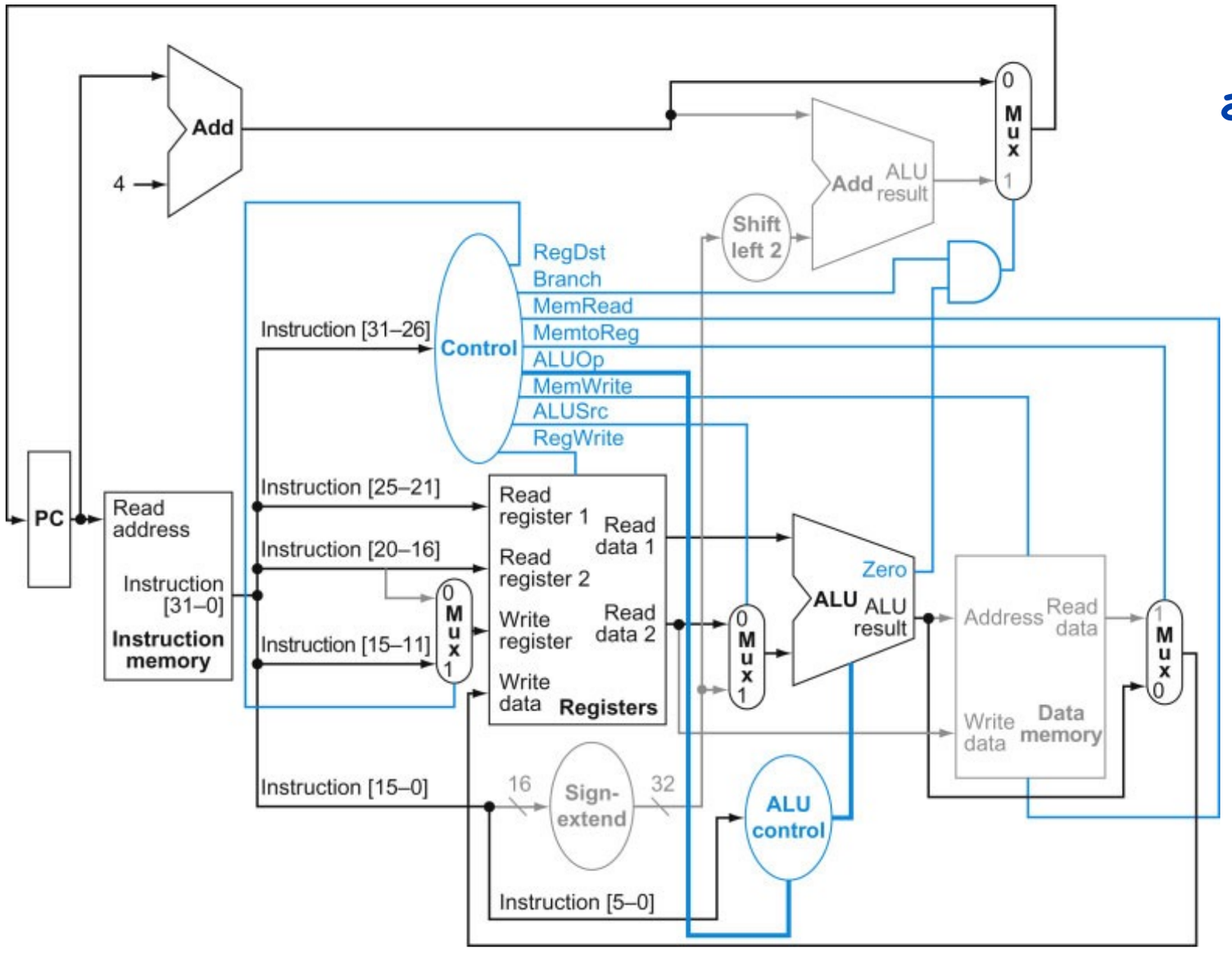


1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. A register ($t2) value is read from the register file.
3. The ALU computes the sum of the value read from the register file and the sign-extended, lower 16 bits of the instruction (**offset**).
4. The sum from the ALU is used as the address for the data memory.
5. The data from the memory unit is written into the register file; the register destination is given by bits 20:16 of the instruction ($t1) .

In a store instruction

- 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.

FIGURE 4.19 The datapath in operation for an **R-type instruction**

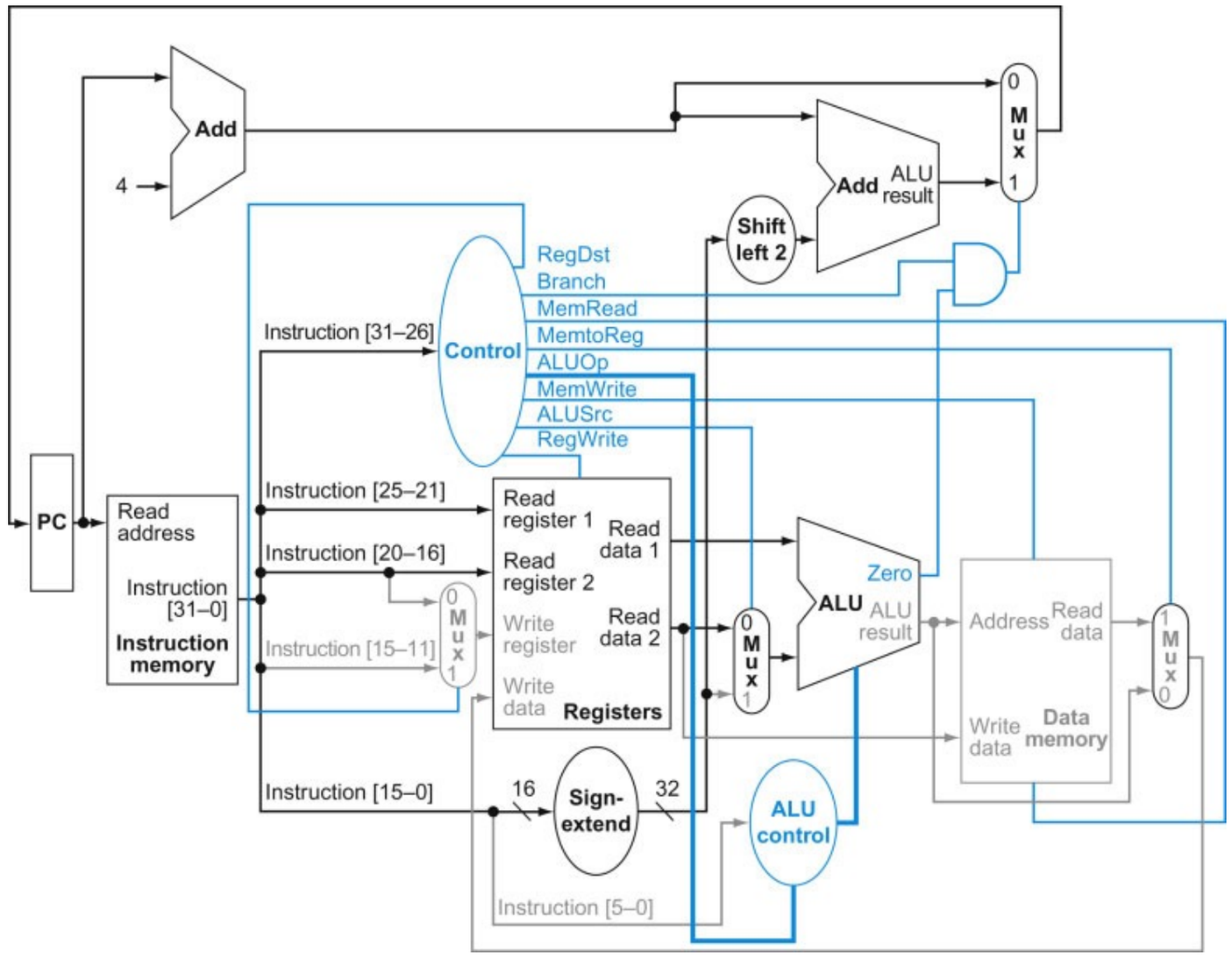



```
add  $t1, $t2, $t3
```

1. The instruction is fetched, and the PC is incremented.
2. Two registers, $t2 and $t3, are read from the register file; also, the main control unit computes the setting of the control lines during this step.
3. The ALU operates on the data read from the register file, using the function code (bits 5:0, which is the funct field, of the instruction) to generate the ALU function.
4. The result from the ALU is written into the register file using bits 15:11 of the instruction to select the destination register ($t1).

FIGURE 4.21 The datapath in operation for a **branch-on-equal** instruction.

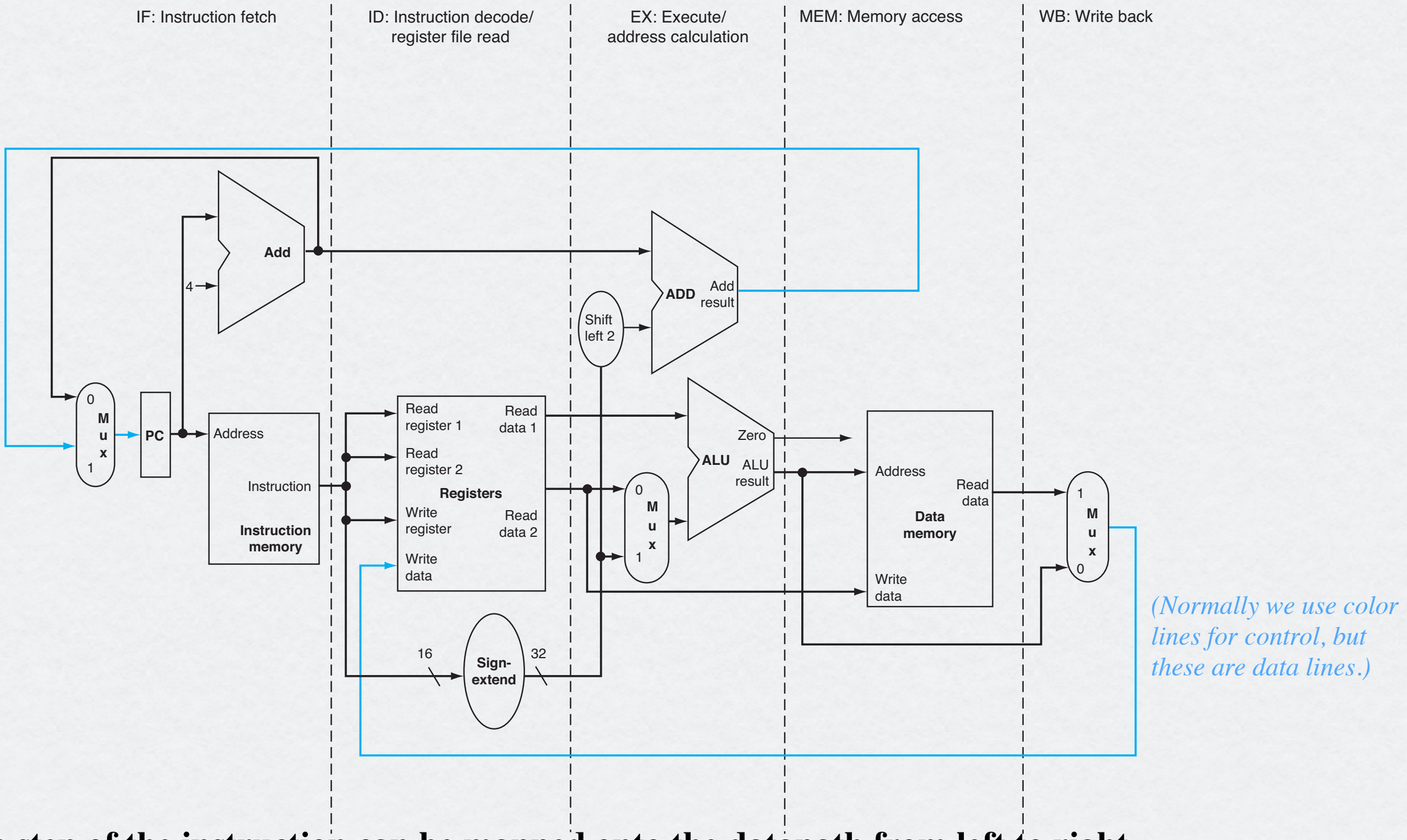
`beq $t1, $t2, offset`



The control lines, datapath units, and connections that are active are highlighted.

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. Two registers, $t1 and $t2, are read from the register file.
3. The ALU performs a subtract on the data values read from the register file. The value of PC + 4 is added to the sign-extended, lower 16 bits of the instruction (offset) shifted left by two; the result is the branch target address.
4. The Zero result from the ALU is used to decide which adder result to store into the PC.

Pipelined Data-path and Control

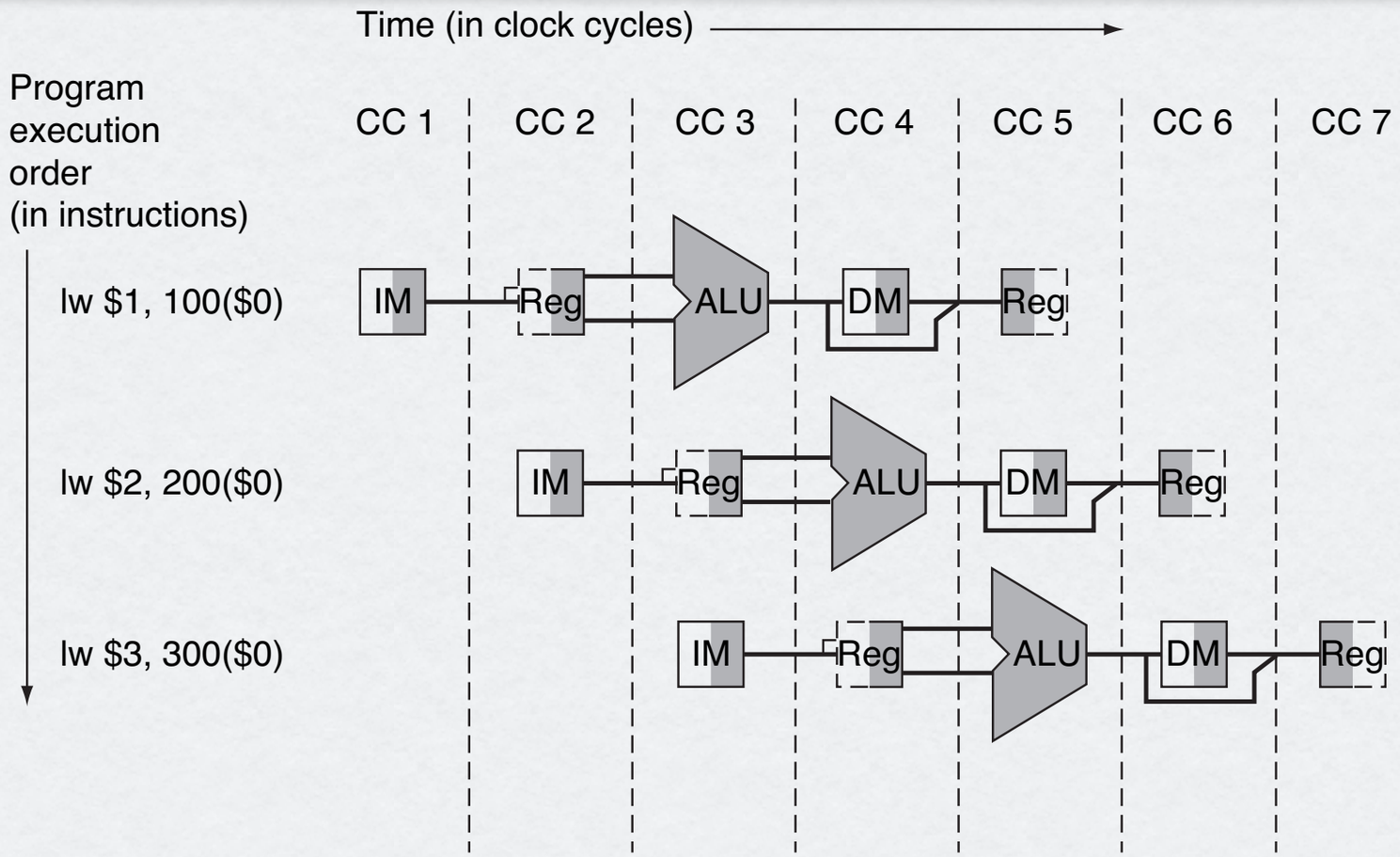


(Normally we use color lines for control, but these are data lines.)

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.

Instructions being executed



This figure pretends that each instruction has its own datapath, and shades each portion according to use.

Each stage is labeled by the physical resource used in that stage, corresponding to the portions of the datapath in Figure 4.33 (previous slide). *IM* represents the Instruction Memory and the PC in the instruction fetch stage, *Reg* stands for the register file and sign extender in the **instruction decode/register file read stage (ID)**, DM stands for **Data Memory**

To maintain proper time order, this stylized datapath breaks the register file into two logical parts: registers read during register fetch (ID) and registers written during write back (WB).

This dual use is represented by drawing the *unshaded left half of the register file* using dashed lines in the ID stage, when it is not being written, and the *unshaded right half in dashed lines in the WB stage*, when it is not being read. As before, we assume the register file is written in the first half of the clock cycle and the register file is read during the second half.