CSC258: Computer Organization

Assembly and Machine Code

Reading Check

1. List the three machine code instruction formats. What is each used for?

2. How is "Make the common case fast" reflected in the MIPS ISA?

3. For "Good design demands good compromises," the text notes that MIPS supports three instruction formats. Why is this a compromise?

Machine Code

This Week's Questions

- What is the difference between assembly and machine code?
- What types of MIPS instructions are there?
  - What is the interface to memory?
- How do we build higher-level constructs like arrays, conditionals, and loops?

What Do Instructions Look Like?

- From the machine's perspective, an instruction is a word of binary data in an easily decoded format.
  - This format is the *machine language*.

- From the programmer's perspective, an instruction is a simple operation, its inputs, and its output.
  - The programmer must know about storage in the processor -- registers.
  - The programmer must know about supported operations
  - This is the *assembly language*.

Two Types of Instruction Sets

- RISC (reduced instruction set computer)
  - Provides very simple (and very fast) instructions
  - Complex instructions are built out of simple ones by the compiler and assembler
- CISC (complex instruction set computer)
  - Very powerful (and perhaps slow) instructions
  - The instruction decoder (and processor) are more complex
  - The compiler has to do a lot of work

MIPS

- MIPS, our ISA, is *register-to-register*.
  - Every operation operates on data in registers.
  - Memory is accessed with *load* and *store*.

- Instructions come in three formats based on their input:
  - Register: the instruction takes 3 registers
  - Immediate: the instruction takes 2 registers and a constant
  - Jump: the instruction takes a large constant

Registers

- MIPS provides 32 registers.
- Several have special values:
  - Register 0 ($zero): value 0 -- always.
  - Registers 2-3 ($v0, $v1): return values
  - Registers 4-7 ($a0-$a3): function arguments
  - Registers 8-15, 24-25 ($t0-$t9): temporaries
  - Registers 16-23 ($s0-$s7): saved temporaries
  - Registers 28-31 ($gp, $sp, $fp, $ra): memory and function support

Quiz-like Question (See Ex 6.7-6.8)

- The user wants two values to be added:
  C = A + B
- The values must be in registers:
  A in $t0
  B in $t1
  C in $t2
- Assembly command:
- Machine code:

Quiz-like Question (See Ex 6.7-6.8)

- The user wants two values to be added:
  C = A + B
- The values must be in registers:
  A in $t0
  B in $t1
  C in $t2
- Assembly command:
  add $t2, $t0, $t1
- Machine code:
  000000 01000 01001 01010 XXXXX 100000

Review: Trace How Add is Executed (Credit: DDCA)

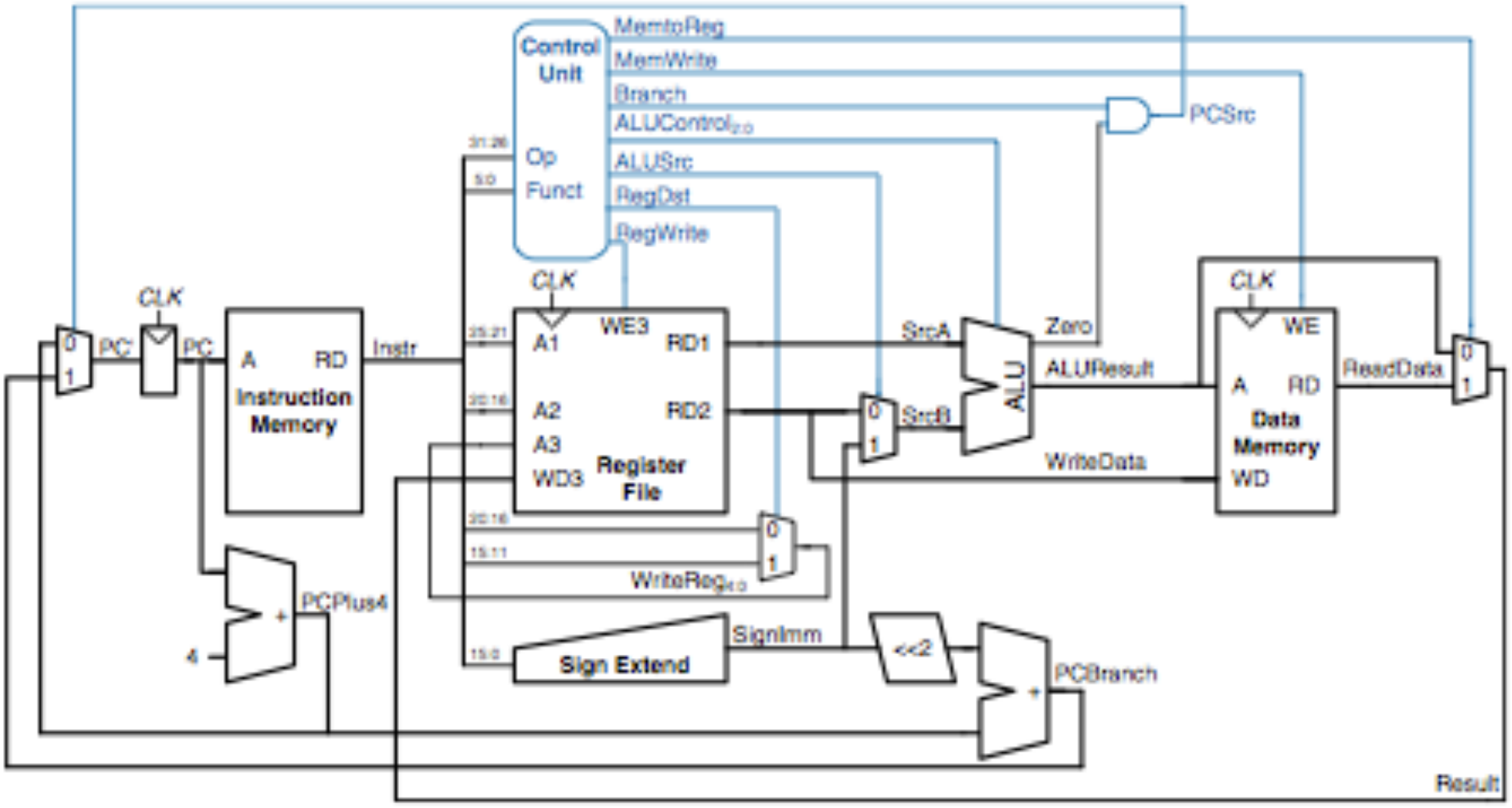


Figure 7.11 Complete single-cycle MIPS processor

Quiz-like Question (See Ex6.7-6.8)

- The user wants two values to be added:
  C = A + 6
- The values must be in registers:
  A in $t0
  C in $t2
- Assembly command:
- Machine code:

Quiz-like Question (See Ex6.7-6.8)

- The user wants two values to be added:
  C = A + 6
- The values must be in registers:
  A in $t0
  C in $t2
- Assembly command:
  addi $t2, $t0, 6
- Machine code:
  001000 01000 01010 0000000000001010

Quiz-like Question (See RG)

- Why do we not have an instruction with two immediate values?
  - Explain why we can't have such an instruction with the existing ISA.
  - Cite the design principles that explain why we don't have such an instruction.

Design Principles

- Simplicity favors regularity
- Make the common case fast
- Smaller is faster
- Good design demands good compromises

Coding Constraints

- The limitations of the encoding put constraints on the code we can write.
  - There is not enough room to encode two immediate values in any of our formats.
  - An extra encoding for an uncommon case is not a good compromise.

Coding Constraints

- The limitations of the encoding put constraints on the code we can write.
  - There is not enough room to encode two immediate values in any of our formats.
  - An extra encoding for an uncommon case is not a good compromise.

- A more common case is using two operands that are in memory. Why don't we allow in-memory instructions?

What if the Data is in Memory?

- The user wants to add two integers *A and *B and store back to the address B.
  - A and B are memory locations!
- MIPS cannot operate directly on values in memory. They must be loaded (or stored). Here is the assembly, assuming A is $t0 and B is $t1.

```
lw $t2, 0($t0)    # Load a word from address A + 0
lw $t3, 0($t1)
add $t3, $t2, $t3
sw $t3, 0($t1)    # Store a word to address B + 0
```

Loading

The load instruction specifies the address to load from and the destination register.

1. The address is sent to memory.
2. The processor "stalls" until the memory system returns the loaded value.

   We have to do this because it takes a while for the result to be returned from memory.
3. The loaded value is moved into the register file.

Cost of Memory Operations

- Foreshadowing: Later this term, we'll consider *caching*, which is a way to reduce the cost of memory operations.

- In a modern processor, a memory operation can take several hundred times longer than a register-to-register operation.
  - This leads to an issue called the *memory wall*.
  - Processors are constrained by the distance to memory, rather than instruction execution speed.

Back to the Question ….

A more common case is using two operands that are in memory. Why don't we allow in-memory instructions?

- … almost all the design principles say we should do this.
- We want the common case to be fast, and loads are not fast.
- Adding memory addressing would likely increase the number of instruction formats and would increase their complexity.

Storing

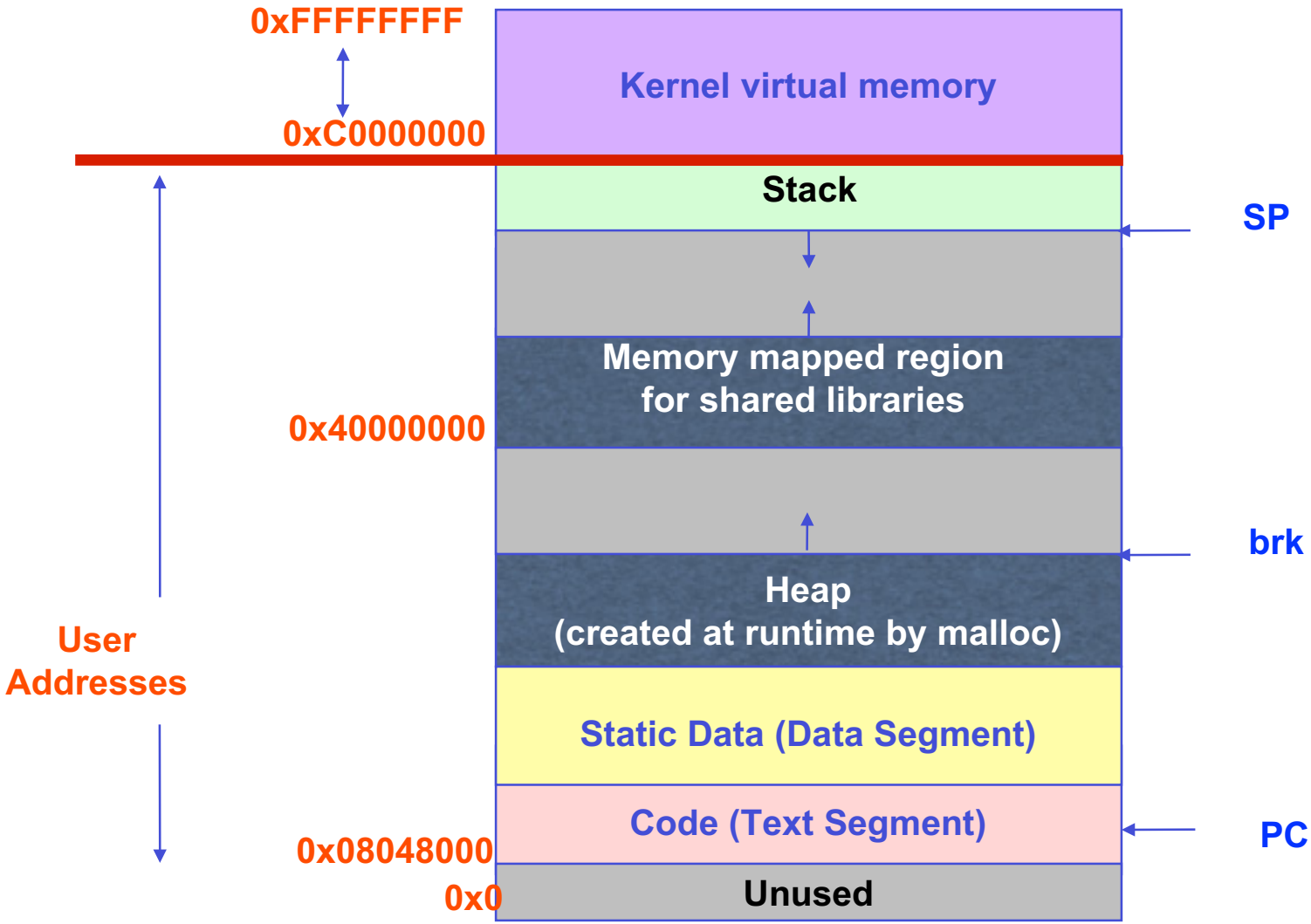
- The store instruction specifies the register with the value to be stored and an address to store into

- ... once those addresses arrive at the memory unit, we're done -- without waiting!
  - The memory system will perform the operation, and the processor can continue to execute, since a store operation does not return a value

The Stack and Heap

- Memory contains two structures that the program can use for extra storage: the stack and heap.
- The pointer to the top of the stack is contained in a hardware register: the stack pointer (SP)
- It is the program's responsibility to manage the stack.
  - *Whenever stack space is allocated, it should be deallocated later by the same function*
- The operating system manages heap allocation.
  - Whenever you *malloc*, you're getting heap space.

Memory Layout (Linux, x86)



Control: Ifs and Loops

- The textbook has good examples of the basic control structures (if statements and loops) and array accesses

- I also recommend using a flowchart to diagram what you want the code to do.
  - The translation from flowchart to code (and vice versa) is fairly straightforward.

Quiz-like Question (See RG)

```
  bne $s3,$s4,L1
  add $s0,$s1,$s2
  j L2
L1:
  sub $s0,$s0,$s3
L2:
```

- What high-level construct does this implement?

Quiz-like Question (See RG)

```
  add $s1,$0,$0
  addi $s0,$0,0
  addi $t0, $0, 10
L1:
  beq $s0, $t0, L2
  add $s1,$s1,$s0
  addi $s0, $s0, 1
  j L1
L2:
```

- What high-level construct does this implement?

Exam-like Question (See Q2)

- Write MIPS assembly code that finds the largest value in an array.
- Assume that the array's base address and the number of array elements are in $a0 and $a1, respectively