CS 4240: Compilers

Lecture 9: MIPS Processor Architecture, SPIM simulator

Tom Conte (Guest Lecturer)

February 11, 2019

REMINDERS

- Project 1 is due by 11:59pm on Wednesday, 2/13/19 on Canvas
  - Must be submitted as zip file including instructions on how to build and run your project
  - 100 points total, with an extra credit option for 15 points
    - Extra credit relates to use of copy propagation
  - 5% of course grade

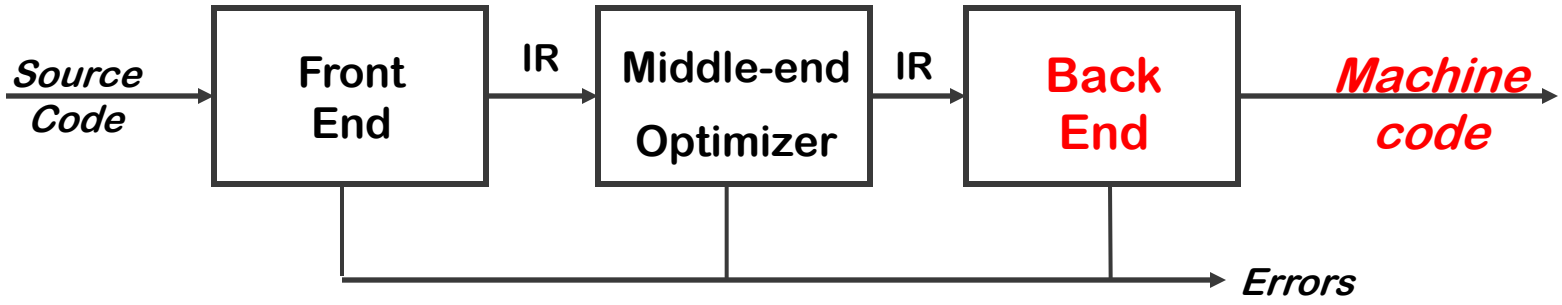
- MIDTERM EXAM: Wednesday, March 13, 4:30pm - 5:45pm

- FINAL EXAM: Wednesday, May 1, 2:40 pm - 5:30 pm

Acknowledgments

- Slides from the following sources
  - http://web.engr.oregonstate.edu/~walkiner/cs271-wi13/slides/02-MIPSArchitecture.pdf
  - https://www.cs.utah.edu/~rajeev/cs3810/slides/3810-06.ppt
  - https://courses.cs.washington.edu/courses/cse410/08sp/notes/spim_tutorial/SpimTutorial.ppt
  - https://www2.engr.arizona.edu/~ece369/Resources/spim/QtSPIM_examples.pdf

Traditional Three-pass Compiler



Role of the "back end"

- Input: optimized IR from the "middle end"
- Output: generate optimized machine code for a given target architecture, while performing
  - Instruction selection (Chapter 11)
  - Register allocation (Chapter 13)
  - Instruction scheduling (Chapter 12)
- Today's lecture will focus on the MIPS processor architecture (the target for your next class project), and its SPIM simulator

CISC vs. RISC

- Intel's IA-32 instruction set has evolved over 20 years – old features are preserved for software compatibility
- Numerous complex instructions – complicates hardware design (Complex Instruction Set Computer – CISC)
- Instructions have different sizes, operands can be in registers or memory, only 8 general-purpose registers, one of the operands is over-written
- RISC instructions are more amenable to high performance (clock speed and parallelism) – MIPS is one of the earliest examples of a RISC processor

MIPS Architecture

- 32-bit RISC Processor
- 32 32-bit Registers, $0..$31
- Pipelined Execution of Instructions
- All instructions are 32-bits
- Most Instructions executed in one clock cycle
- 2^{30} 32-bit memory words
- Byte addressable memory
  - A 32-bit word contains four bytes
  - To address the next word of memory add 4

How a computer executes a program

Fetch-decode-execute cycle (FDX)

1. **fetch** the next instruction from memory
2. **decode** the instruction
3. **execute** the instruction

Decode determines:

- operation to execute
- arguments to use
- where the result will be stored

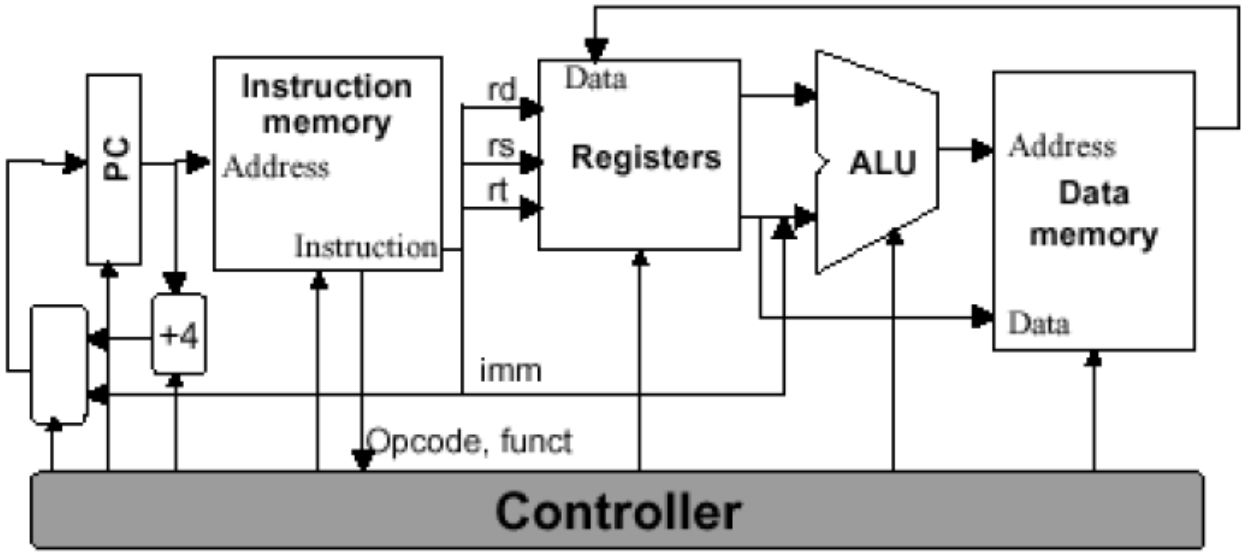
Execute:

- performs the operation
- determines next instruction to fetch (by default, next one)

Major components of the datapath:

- program counter (PC)
- instruction register (IR)
- register file
- arithmetic and logic unit (ALU)
- memory

Control unit



Five-stage pipeline

- Note: Delayed branches are not modeled in SPIM unless use the –bare flag

Memory: text segment vs. data segment

In MIPS, programs are separated from data in memory

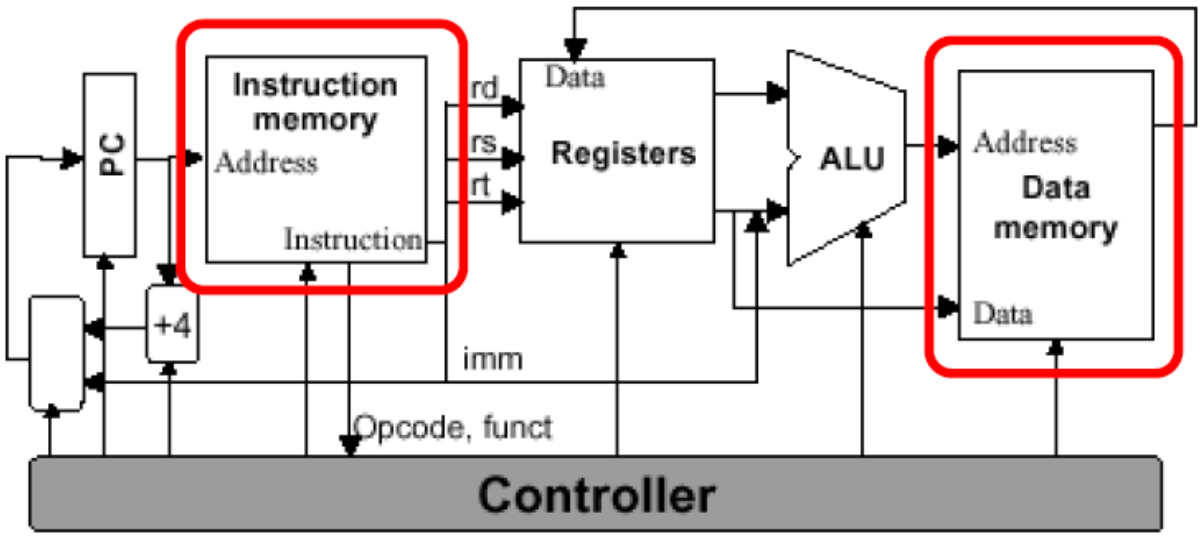
Text segment

- "instruction memory"
- part of memory that stores the program (machine code)
- **read only**

Data segment

- "data memory"
- part of memory that stores data manipulated by program
- **read/write**

Memory: text segment vs. data segment



Distinction may or may not be reflected in the hardware:

- von Neumann architecture – single, shared memory
- Harvard architecture – physically separate memories

Memory addressing in MIPS

For reading/writing the data segment

Base address plus displacement

Memory address computed as base+offset:

- base is obtained from a register
- offset is given directly as an integer

Load word (read word from memory into register):

```
lw $t1,8($t2)   ⇒   $t1 := Memory[$t2+8]
```

Store word (write word from register into memory):

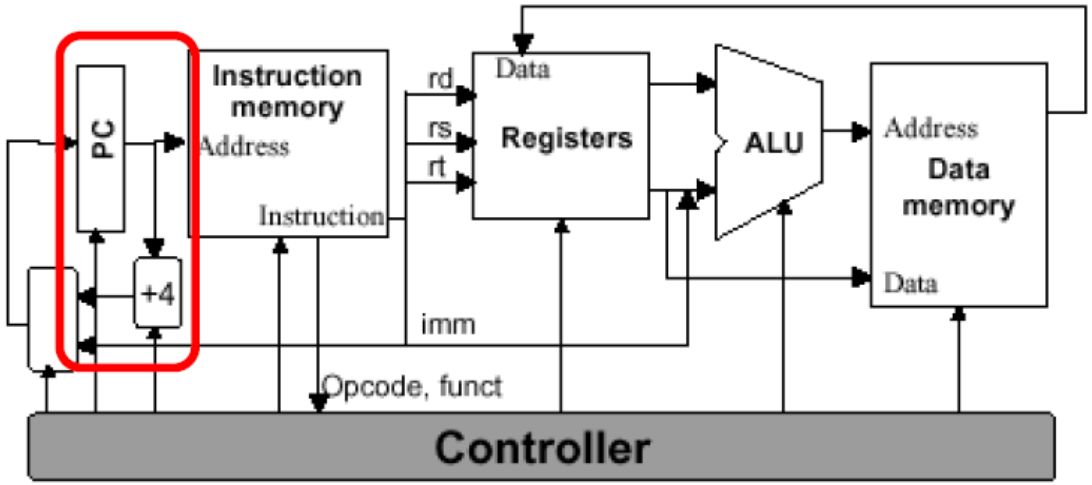
```
sw $t1,4($t2)   ⇒   Memory[$t2+4] := $t1
```

We'll talk about addressing in the text segment later

Program counter (PC)

Program: a sequence of machine instructions in the text segment

```
0x8d0b0000
0x8d0c0004
0x016c5020
0xad0a0008
0x21080004
0x2129ffff
0x1d20fff9
```



Program counter

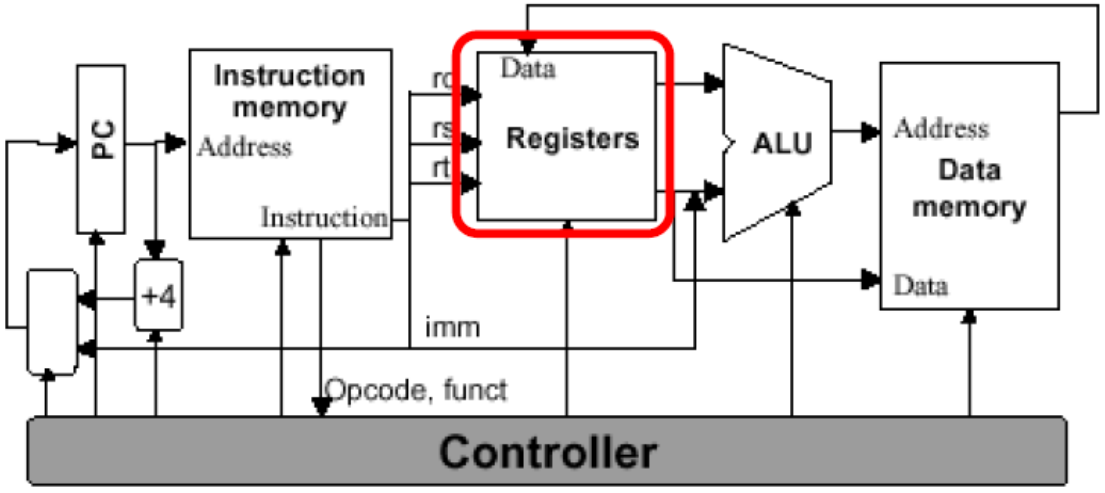
Register that stores the **address** of the next instruction to fetch

- also called the instruction pointer (IP)

Register file

Register: component that stores a 32-bit value

MIPS register file contains 32 registers



Register names and conventions

Number	Name	Usage	Preserved?
$0	$zero	constant 0x00000000	N/A
$1	$at	assembler temporary	✗
$2–$3	$v0–$v1	function return values	✗
$4–$7	$a0–$a3	function arguments	✗
$8–$15	$t0–$t7	temporaries	✗
$16–$23	$s0–$s7	saved temporaries	✓
$24–$25	$t8–$t9	more temporaries	✗
$26–$27	$k0–$k1	reserved for OS kernel	N/A
$28	$gp	global pointer	✓
$29	$sp	stack pointer	✓
$30	$fp	frame pointer	✓
$31	$ra	return address	✓

Arithmetic and logic unit (ALU)

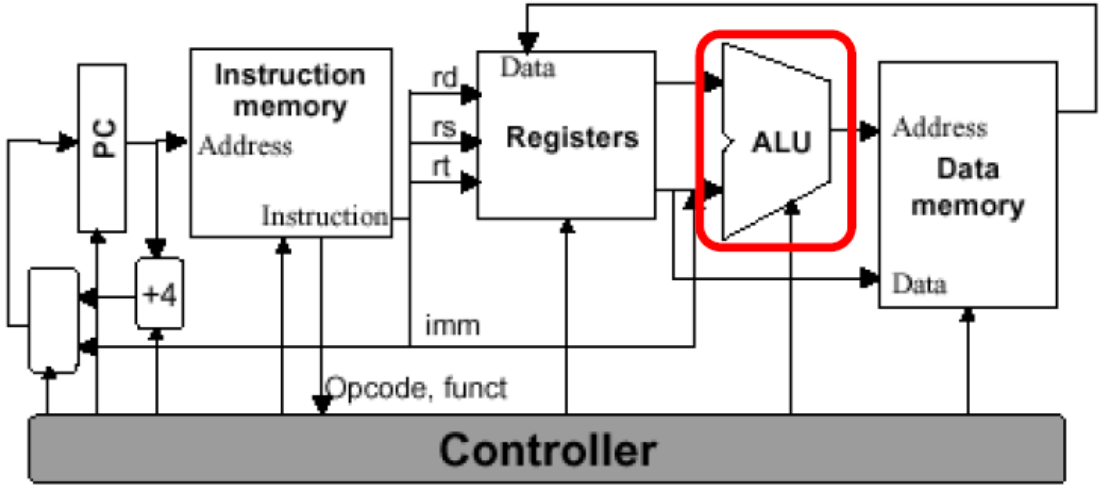
Implements binary arithmetic and logic operations

Inputs:

- operands – 2 $\times$ 32-bit
- operation – control signal

Outputs:

- result – 1 $\times$ 64-bit
  (usually just use 32 bits of this)
- status – condition signals

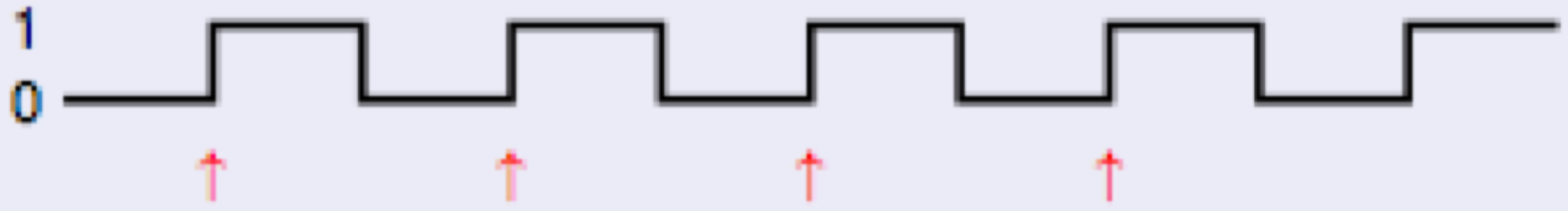


Clock signal

Each component is implemented as an electrical circuit

- when inputs change, outputs change – not instantaneous!
- clock signal ensures we don't use outputs until ready

Clock generator produces the clock signal



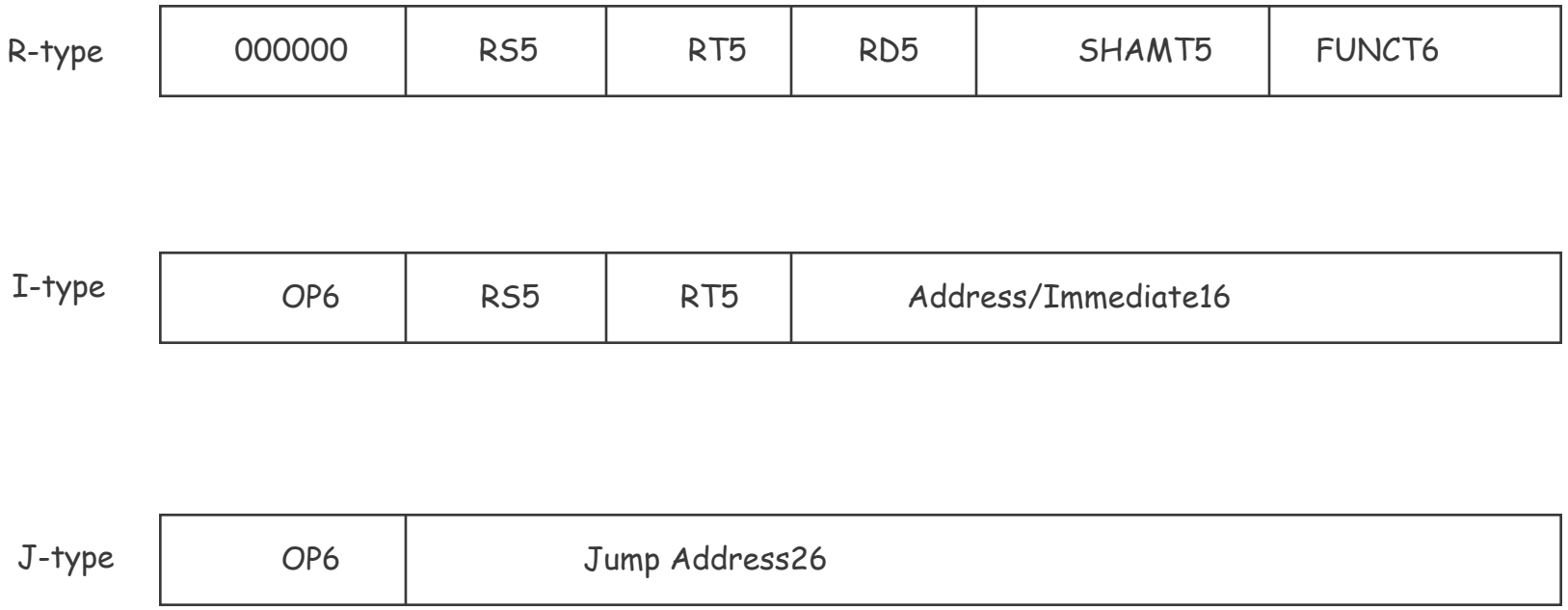
- synchronizes the components in the data path
- the faster the clock, the faster the program executes
  - clock rate is limited by the slowest component!

MIPS Instruction Formats

- R format - Uses three register operands
  - Used by all arithmetic and logical instructions
- I format - Uses two register operands and an address/immediate value
  - Used by load and store instructions
  - Used by arithmetic and logical instructions with a constant
- J format - Contains a jump address
  - Used by Jump instructions

MIPS Instruction Formats

» 32-bit Instruction Formats R, I and J



Total encodings: 64 (R-type) + 64-1 (I and J type) or 127

MIPS Instruction Set

- Only Load instruction can read an operand from memory
- Only Store instruction can write an operand to memory
- Typcial RISC calculations require
  - Load(s) to get operands from memory into registers
  - Calculations performed only on values in registers
  - Store(s) to write result from register back to memory

MIPS Addressing Modes

- Register - Uses value in register as operand
  - Example $2 - Uses register 2 as operand
- Direct Address - Uses value stored in memory at given address
  - Example 100 - Uses value stored at location 100 in memory as operand
- Register Indirect Address - Uses value in register as address of operand in memory
  - Example ($3) - Uses value in register 3 as address of memory operand

MIPS Addressing Modes

- Indexed Address - Adds address field and register value and uses this as address of operand in memory
  - Example 100($2) - Adds 100 to the value in register 2 and reads the operand from the resulting memory address
  - Used for array X[I] operations, the array index is normally in the register and the address field is the first location in the array

MIPS Addressing Modes

- Immediate Addressing - Uses constant value contained in instruction
  - Example addi $1,$2,4 - adds constant 4 to register 2 and stores result in register 1
  - Used for constants in programs
- PC relative - Address from instruction is added to the current value in the Program Counter
  - Example J 4 - Jumps to address PC+4
  - Saves address bits in jump instructions

MIPS Assembly Language Examples

- ADD $1,$2,$3
  - Register 1 = Register 2 + Register 3
- SUB $1,$2,$3
  - Register 1 = Register 2 - Register 3
- AND $1,$2,$3
  - Register 1 = Register 2 AND Register 3
- ADDI $1,$2,10
  - Register 1 = Register 2 + 10
- SLL $1, $2, 10
  - Register 1 = Register 2 shifted left 10 bits

MIPS Assembly Language Examples

- LW $1,100
  - Register 1 = Contents of memory location 100
  - Used to load registers
- SW $1,100
  - Contents of memory location 100 = Register 1
  - Used to save registers
- LUI $1,100
  - Register 1 upper 16-bits = 100
  - Lower 16-bits are set to all 0's
  - Used to load a constant in the upper 16-bits
  - Other constants in instructions are only 16-bits, so this instruction is used when a constant larger than 16-bits is required for the operation

  Constant to load : 0xABCD1234

  LUI $t0, 0xABCD

  ADDI $t0, $0, 0x1234

  (Assembler does this if you use LI $t0, 0xABCD1234)

MIPS Assembly Language Examples

- J 100
  - Jumps to PC+100
- JAL 100
  - Save PC in $31 and Jumps to PC+100
  - Used for subroutine calls
- JR $31
  - Jumps to address in register 31
  - Used for subroutine returns

MIPS Assembly Language Examples

- BEQ $1, $2, 100
  - If register 1 equal to register 2 jump to PC+100
  - Used for Assembly Language If statements
- BNE $1, $2, 100
  - If register 1 not equal to register 2 jump to PC+100
  - Used for Assembly Language If statements

Also
BGTZ, $1, 100
BLEZ $1, 100

MIPS Assembly Language Examples

- SLT $1,$2,$3
  - If register 2 is less than register 3 then register 1=1 else register 1=0
  - In an assembly language flag a "1" means true and a "0" means false
  - Used in conjunction with Bxx instruction to implement any arithmetic comparision
  - Required for more complex assembly language If statements

  Also

  SLTI $1, $2, Imm

MIPS Labels

- Instead of absolute memory addresses symbolic labels are used to indicate memory addresses in assembly language
- Assembly Language Programs are easier to modify and are easier to understand when labels are used
  - Examples
  - Variable X is stored a location 123 in memory - Use label X instead of 123 in programs.
  - Location LOOP_TOP is address 250 in a program - Use label LOOP_TOP instead of jump address 250 in programs

MIPS Program Examples
A = B + C;

```
LW  $2, B           ;Register 2 = value of B
LW  $3, C           ;Register 3 = value of C
ADD $4, $2, $3      ;Register 4 = B+C
SW  $4, A           ; A = B + C
```

MIPS Assembly Language Label Examples

N: .WORD 0

- Like declaring an Integer, N, in a HLL
- Sets up N as a Label that points to a 32-bit data value
- Initial value is set to 0

LOOP: ADD $a0,$a0,$a1

- Sets up LOOP as a Label that points to the Add instruction
- Can jump to LOOP (i.e. J LOOP)

MIPS Assembler Directives

- Assembler directives are commands for the assembler that do not generate machine instructions
- Assembler directives are also called pseudo ops
- Used to set up data and instruction areas

- Some pseudo instructions:
  - LI $t1, 0xABCD1234 // Load immediate
  - LA $t1, 0xABCD1234 // Load address

MIPS Assembler Directive Examples

- .DATA
  - The following lines are constant or data values
- .WORD *value*
  - Reserve a 32-bit word in memory for a variable
- .ASCIIZ "*string*"
  - Place a string in memory using the ASCII character code, end with a null
- .TEXT
  - The following lines are instructions
- .GLOBL *label*
  - Indicates to the linker that a label is a global label

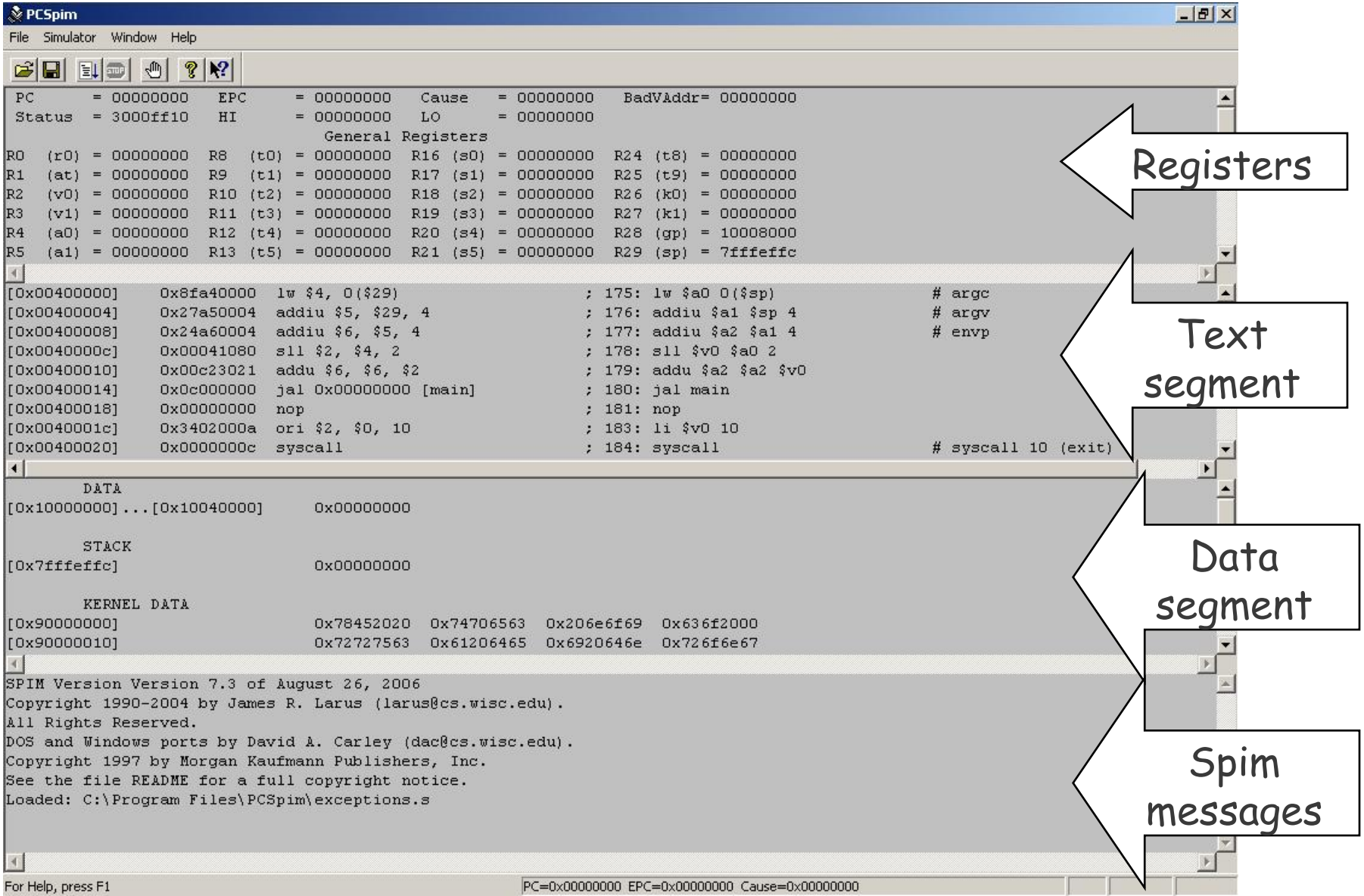
SPIM Simulator Windows

- Text Window
  - Contains Assembled Machine Instructions
  - Use to Obtain Program and Breakpoint Addresses
- Register Window
  - Displays value of machine registers when program is run
  - Check after stopping at a breakpoint to aid debug
- Data Window
  - Displays values of data in memory
- Console Window
  - Used for Program Input and Output
- Session Window
  - Displays Assembly Errors

SPIM

- SPIM is a simulator that reads in an assembly program and models its behavior on a MIPS processor
- Note that a “MIPS add instruction” will eventually be converted to an add instruction for the host computer’s architecture – this translation happens under the hood
- To simplify the programmer’s task, it accepts pseudo-instructions, large constants, constants in decimal/hex formats, labels, etc.
- The simulator allows us to inspect register/memory values to confirm that our program is behaving correctly

SPIM Environment



Example

This simple program will run on SPIM (a “main” label is introduced so SPIM knows where to start)

```
main:
    addi   $t0, $zero, 5
    addi   $t1, $zero, 7
    add    $t2, $t0, $t1
```

If we inspect the contents of $t2, we’ll find the number 12

User Interface

```
> spim

(spim)  read “add.s”
(spim)  run
(spim)  print $10
  Reg 10 = 0x0000000c (12)
(spim)  reinitialize
(spim)  read “add.s”
(spim)  step
(spim)  print $8
  Reg 8 = 0x00000005  (5)
(spim)  print $9
  Reg 9 = 0x00000000  (0)
(spim)  step
(spim)  print $9
  Reg 9 = 0x00000007  (7)
(spim)  exit
```

File add.s

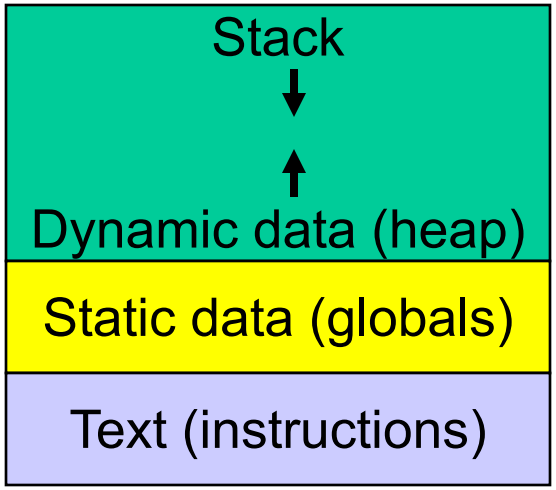
```
main:
    addi   $t0, $zero, 5
    addi   $t1, $zero, 7
    add    $t2, $t0, $t1
```

Directives (Text segment)

File add.s

```
.text
.globl main
main:
    addi   $t0, $zero, 5
    addi   $t1, $zero, 7
    add    $t2, $t0, $t1
    …
    jal      swap_proc
    jr       $ra

.globl swap_proc
swap_proc:
     …
```

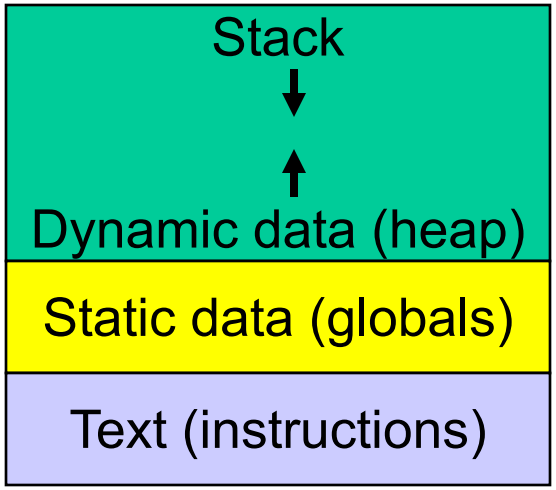


This function is visible to other files

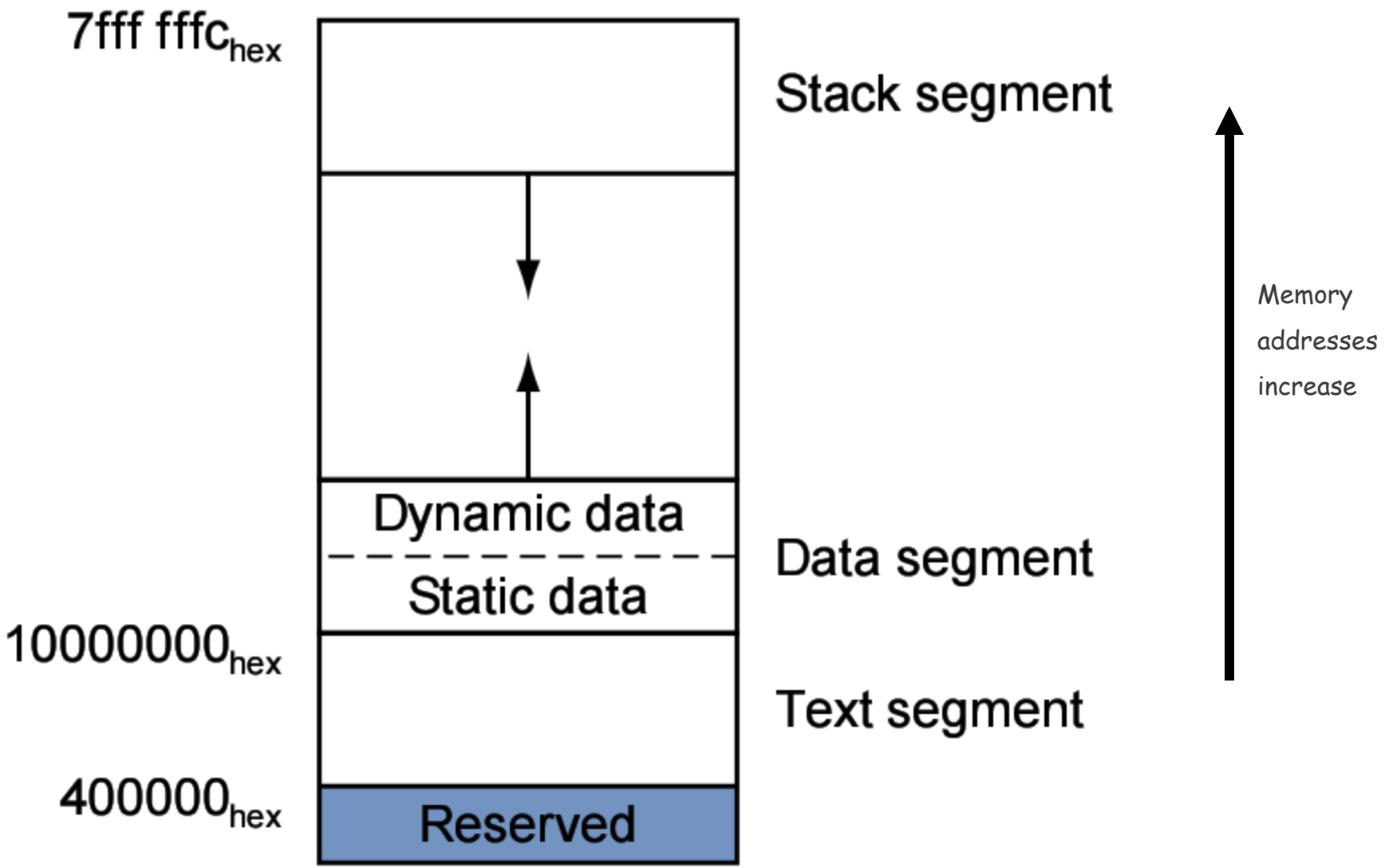
Directives (Data segment with labels)

File add.s

```
.data
in1   .word 5
in2   .word 7
c1    .byte  25
str   .asciiz  “the answer is”
.text
.globl main
main:
    lw      $t0, in1
    lw      $t1, in2
    add    $t2, $t0, $t1
    …
    jal      swap_proc
    jr      $ra
```



Memory layout



Endian-ness

Two major formats for transferring values between registers and memory

Memory: low address 45 7b 87 7f high address

Little-endian register: the first byte read goes in the low end of the register

Register: 7f 87 7b 45

Most-significant bit ↗ ↖ Least-significant bit

Big-endian register: the first byte read goes in the big end of the register

Register: 45 7b 87 7f

Most-significant bit ↗ ↖ Least-significant bit

System Calls

- SPIM provides some OS services: most useful are operations for I/O: read, write, file open, file close
- The arguments for the syscall are placed in $a0-$a3
- The type of syscall is identified by placing the appropriate number in $v0 – 1 for print_int, 4 for print_string, 5 for read_int, etc.
- $v0 is also used for the syscall's return value

Example Print Routine

```
.data
   str:     .asciiz   “the answer is ”
.text
   li       $v0, 4              # load immediate; 4 is the code for print_string
   la       $a0, str            #  the print_string syscall expects the string
                                #  address as the argument; la is the instruction
                                #  to load the address of the operand (str)
   syscall                      #  SPIM will now invoke syscall-4
   li       $v0, 1              #  syscall-1 corresponds to print_int
   li       $a0, 5              #  print_int expects the integer as its argument
   syscall                      #  SPIM will now invoke syscall-1
```

System calls

Service	System Call Code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		

System calls – print_str

```
        .data
str:    .asciiz “Hello World”

        .text
        .globl   main
main:
        li   $v0,  4            # code for print_str
        la  $a0,  str           # argument
        syscall                 # executes print_str
```

System calls – read _int

```
        .data
num:    .space 4

        .text
        .globl   main
main:
        li  $v0,  5             # code for read_int
        syscall                 # executes read_int
                                # return value is stored in $v0
        la   $t0, num           # load address of num to $t0
        sw  $v0, 0($t0)         # store the number in num
```

Branching

x ← read_int

y ← read_int

if x == y

then print “Equal”

else print “Not equal”

Branching

```
        .text
        .globl   main
main:
        li   $v0,  5

        syscall
        move  $s0, $v0

        li   $v0,  5

        syscall
        move  $s1, $v0

        beq  $s0, $s1, printEq

        bne  $s0, $s0, printNe
```

```
printEq:
        la  $a0,  strEq
        j  print

printNe:
        la  $a0, strNe
        j  print

print:
        li   $v0,  4
        syscall

.data
strEq:    .asciiz “Equal”
strNe:    .asciiz “Not equal”
```

Looping

```
x ← read_int

counter ← 0
total   ← 0

do
    counter ← counter + 1
    total   ← total + counter
until counter == x

print total
```

Looping

```
    .text
    .globl main
main:
    li   $v0,  5
    syscall

    move  $t0, $v0

    # $t0 is the original value

    li      $t1, 0    # counter
    li      $t2, 0  # sum
```

```
loop:
    addi $t1, $t1, 1
    add  $t2, $t2, $t1

    beq  $t0, $t1, done
    j loop

done:
    li   $v0,  1     # print_int
    move $a0, $t2
    syscall
```