ECE 550D
Fundamentals of Computer Systems and Engineering

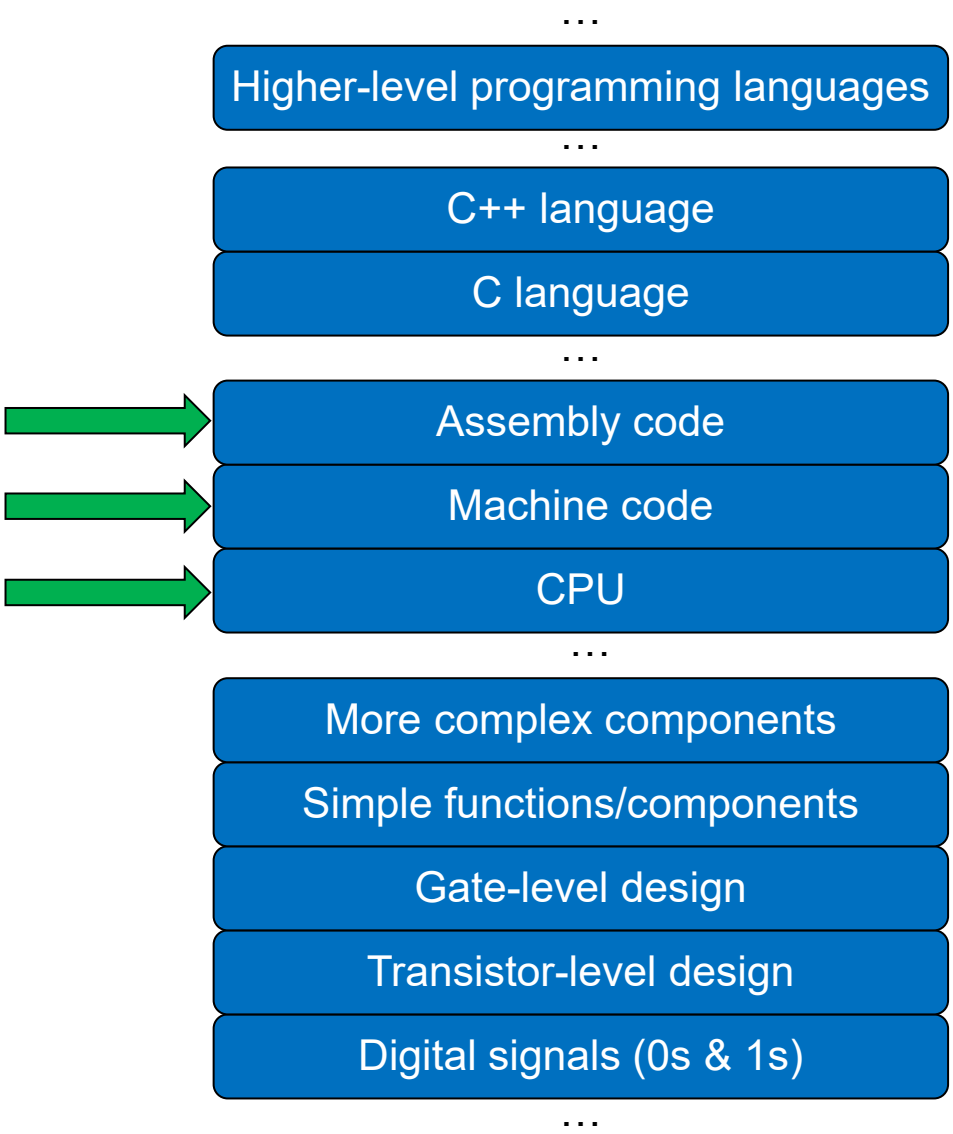
Fall 2025

Instruction Set Architecture (ISA)

Yiran Chen and Hai "Helen" Li
Duke University

Slides are derived from work by
Rabih Younes, John Board, Andrew Hilton, and Tyler Bletsch (Duke)

Running Software



Running Software

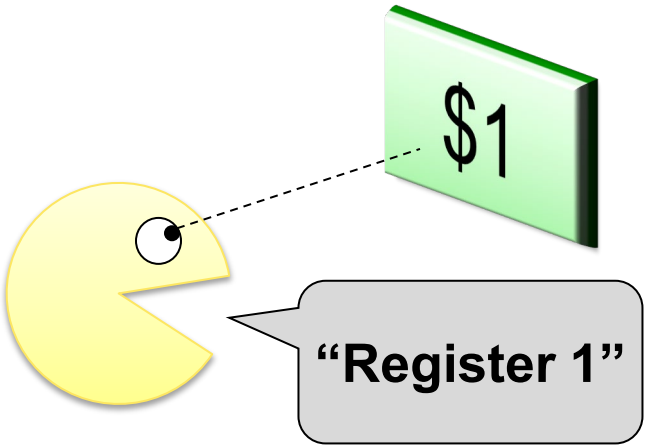
- Software runs on general-purpose hardware (compared to application-specific hardware)
  - The Central Processing Unit (CPU)
    - Also called "processor"
- High-level software is compiled/interpreted into assembly (low-level software/code)
- Assembler then converts assembly into machine code (1s and 0s)

- → CPU design and machine code instructions should be designed to work together
  - CPU should be able to accommodate every machine code instruction
  - Machine code instructions should be able to run on the CPU hardware
    - ~1 instruction per clock cycle (more on this later)
  - The set of instructions we're able to run on a CPU is called the "**instruction set architecture**" (ISA)
    - It is the interface between software (code) and hardware (CPU)

First, Let's See What Assembly Is

- Assembly programming:
  - 1 machine instruction at a time
  - Still in "human readable form", e.g.:
    - `add $1, $2, $3`

How to say "$1" out loud:



  - Much "lower-level" than any other programming language
    - Uses registers in regfile (vs unlimited variables in higher languages)

Registers

- There are 2 places where processors can store data:
  - Registers (saw these -- sort of):
    - In processor
    - Few of them (e.g., 32)
    - Very fast
  - Main memory (later):
    - I.e., RAM
    - Outside of processor
    - Huge (e.g., 8 GB)
    - Much slower than registers

- For now: think of registers like "variables"
  - But we have a limited number of them
  - E.g., $1 = $2 + $3
    - much like s = a + b

Simple Assembly Code Example

```
// silly C code

int sum, temp, x, y;
while (true){
    temp = x + y;
    sum = sum + temp;
}
```

```
// equivalent assembly code (for MIPS*)

loop:   lw $1, Memory[1004]
        lw $2, Memory[1008]
        add $3, $1, $2
        add $4, $4, $3
        j loop
```

Memory references don't quite work like this…we'll correct this later.

OK, so what does this assembly code mean?
Let's dig into each line …

*MIPS: Microprocessor without Interlocked Pipelined Stages

Simple Assembly Code Example

```
loop:   lw $1, Memory[1004]
        lw $2, Memory[1008]
        add $3, $1, $2
        add $4, $4, $3
        j loop
```

```
// silly C code

int sum, temp, x, y;
while (true){
     temp = x + y;
     sum = sum + temp;
}
```

"loop:" = line label (in case we need to refer to this instruction later)
lw = "load word" = read a word (32 bits) from memory
$1 = "register 1" → put result read from memory into register 1
Memory[1004] = address in memory to read from (where x lives)

Note: almost all MIPS instructions put destination (where result gets written) first (in this case, $1)

Simple Assembly Code Example

```
loop:   lw $1, Memory[1004]
        lw $2, Memory[1008]
        add $3, $1, $2
        add $4, $4, $3
        j loop
```

```
// silly C code

int sum, temp, x, y;
while (true){
     temp = x + y;
     sum = sum + temp;
}
```

lw = "load word" = read a word (32 bits) from memory
$2 = "register 2" → put result read from memory into register 2
Memory[1008] = address in memory to read from (where y lives)

Simple Assembly Code Example

```
loop:   lw $1, Memory[1004]
        lw $2, Memory[1008]
        add $3, $1, $2
        add $4, $4, $3
        j loop
```

```
// silly C code

int sum, temp, x, y;
while (true){
     temp = x + y;
     sum = sum + temp;
}
```

add $3, $1, $2 = add what's in $1 to what's in $2 and put result in $3

Simple Assembly Code Example

```
loop:   lw $1, Memory[1004]
        lw $2, Memory[1008]
        add $3, $1, $2
        add $4, $4, $3
        j loop
```

```
// silly C code

int sum, temp, x, y;
while (true){
     temp = x + y;
     sum = sum + temp;
}
```

add $4, $4, $3 = add what's in $4 to what's in $3 and put result in $4

Note: this instruction overwrites previous value in $4

Simple Assembly Code Example

```
loop:   lw $1, Memory[1004]
        lw $2, Memory[1008]
        add $3, $1, $2
        add $4, $4, $3
        j loop
```

```
// silly C code

int sum, temp, x, y;
while (true){
     temp = x + y;
     sum = sum + temp;
}
```

j = "jump"
loop = address of instruction at label "loop" (the first lw instruction above)
sets next PC (Program Counter) to the address labeled by "loop"

Note: all other instructions in this code set next PC = PC++ (remember, this code is stored somewhere in memory and that's why we need a PC to go through the lines of code in memory)

Assembly to Machine Code

- Human readable is not (easily) machine executable
  - `add $1, $2, $3`
- Instructions are numbers too!
  - Bit fields (like FP numbers)
- Instruction Format
  - Establishes a mapping from "instruction" to binary values
  - Which bit positions correspond to which parts of the instruction (operation, operands, etc.)

- In MIPS (a 32-bit CPU), each assembly instruction has a unique 32-bit representation:
  - `add $3, $2, $7` ←→ `00000000010001110001100000100000`
  - `lw $8, Mem[1004]` ←→ `10001100000010000000001111101100`
- Assembler does this translation

What Must Be Specified in an Instruction?

- Instruction "opcode"
  - What should this operation do? (add, subtract,…)
- Location of operands and result
  - Registers (which ones?)
  - Memory (which address?)
  - Immediates/constants (what value?)
- Data type and size
  - Usually included in opcode
  - E.g., signed vs. unsigned int (if it matters)
- What instruction comes next?
  - Sequentially, the next instruction, or jump elsewhere (if/else, loops)

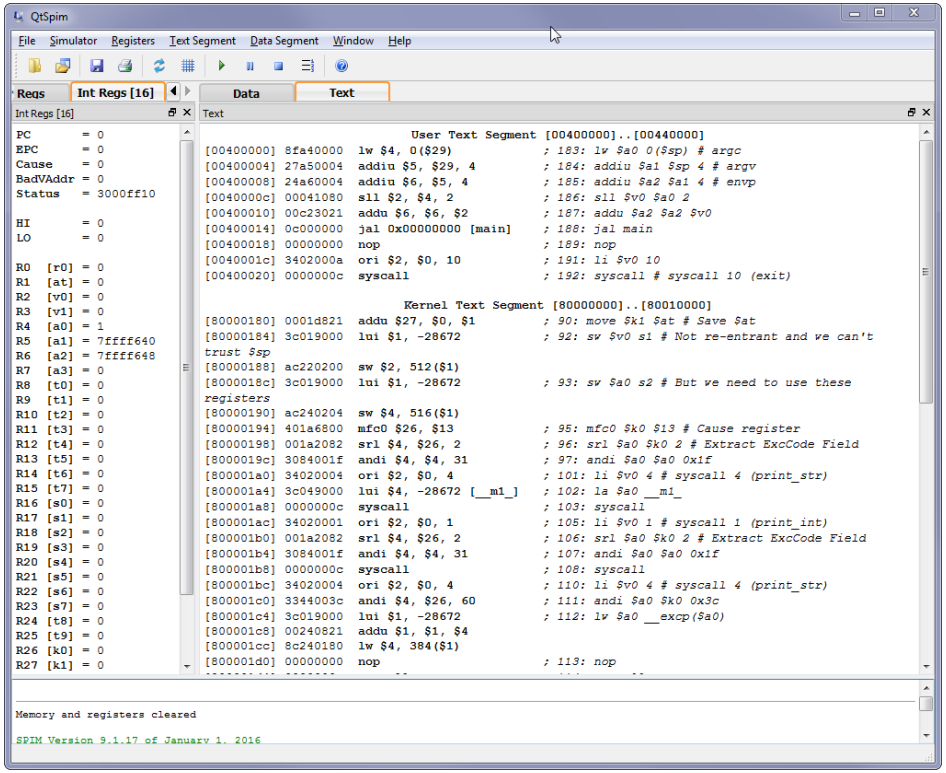
The ISA

- Instruction Set Architecture (ISA)
  - **Contract between hardware and software**
  - Specifies everything hardware and software need to agree on
    - Instruction encoding and effects
    - (lots of other things that won't make sense yet)

- Many different ISAs for different processors
  - x86 and x86_64 (Intel and AMD)
  - POWER (IBM)
  - **MIPS** *(easier to learn; good starting point; we'll use this)*
  - ARM
  - SPARC (Oracle)

But I don't have a MIPS computer?

- You can use SPIM (MIPS simulator)
  - Command-line version: spim
  - Graphical version: qtspim



ISAs: RISC vs CISC

- Two broad categories of ISAs:
  - Complex Instruction Set Computing
    - Came first, days when people always directly wrote assembly
    - Big complex instructions
  - Reduced Instruction Set Computing
    - Goal: make hardware simple and fast
    - Write in high level language, let compiler do the dirty work
      - Rely on compiler to optimize for you

- Note:
  - Sometimes fuzzy: ISAs may have some features of each
  - Common misconception: not about how many different insns!

ISAs: RISC vs CISC

Reduced **I**nstruction **S**et **C**omputing

- Simple, fixed length instruction encoding
- Few **memory addressing modes**
- Instructions have one effect
- "Many" registers (e.g., 32)
- Three-operand arithmetic (dest = src1 op src2)
- **Load-store** ISA

ISAs: RISC vs CISC

- **C**omplex **I**nstruction **S**et **C**omputing
  - Variable length instruction encoding (sometimes quite complex)
  - Many addressing modes, some quite complex
  - Side-effecting and/or complex instructions
  - Few registers (e.g., 8)
  - Various operand models
    - Stack
    - Two-operand (dest = src op dest)
    - Implicit operands
  - Can operate directly on memory
    - Register = Memory op Register
    - Memory = Memory op Register
    - Memory = Memory op Memory

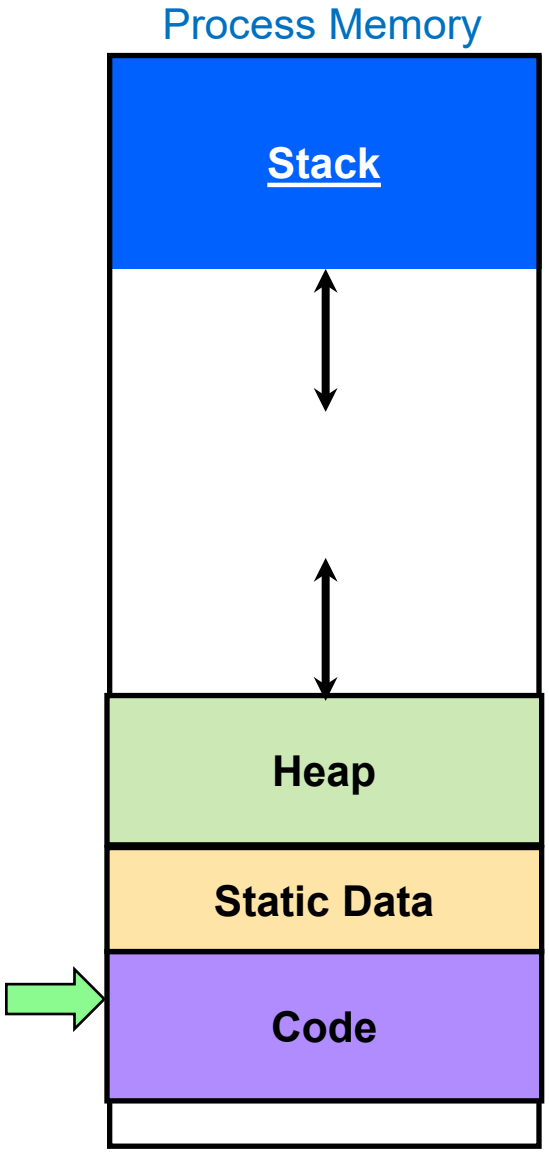
Load-Store ISA

- Load-store ISA:
  - Specific instructions (loads/stores) to access memory
    - Loads read memory (and **only** read memory)
    - Stores write memory (and **only** write memory)
    - Everything else happens in registers

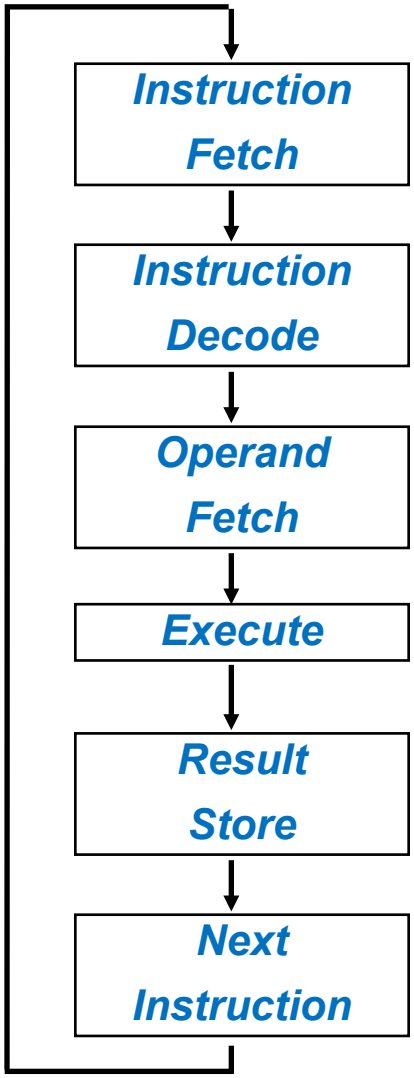
- In contrast with
  - General memory operands ($4 = mem[$5] + $3)
  - Memory/memory operations: mem[$4] = mem[$5] + $3

Process/Program Memory

Instructions and data are stored in memory



How Is Code Executed?

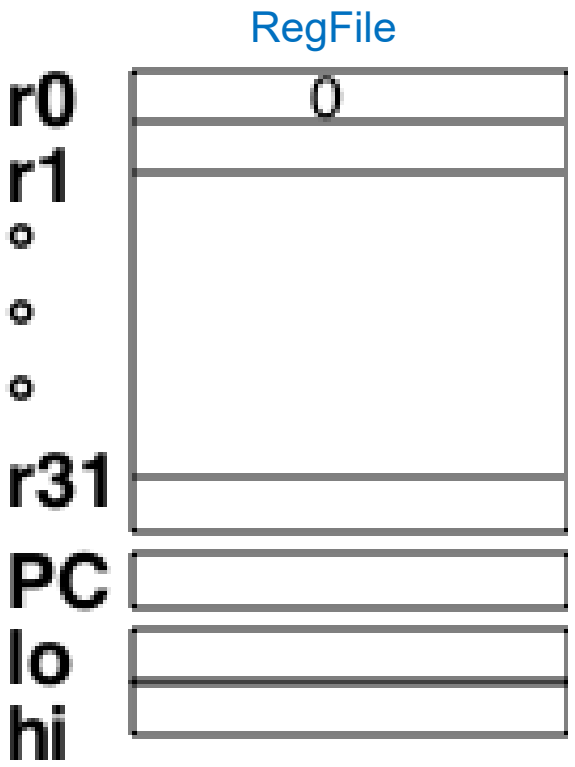


- Instruction Fetch:
  Read instruction bits from memory
- Decode:
  Figure out what those bits mean
- Operand Fetch:
  Read registers (+ mem to get sources)
- Execute:
  Do the actual operation (e.g., add the numbers)
- Result Store:
  Write result to register or memory
- Next Instruction:
  Figure out mem addr of next insn, repeat

Called Von Neumann model

MIPS Registers

- Recall: registers (in regfile)
  - Fast
  - In CPU
  - Directly compute on them
- In MIPS:
  - 32 x 32-bit general-purpose int regs
    - With R0 = 0
  - + floating-point registers
  - + special purpose registers, such as
    - PC = Address of next insn



Assembly Code Execution Example

- What is the simplest computation we might do?
  - Add two numbers:

x = a + b;

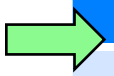
add $1, $2, $3

"Add $2 + $3, and store it in $1"

Note: when writing assembly, basically, pick reg for a, reg for b, reg for x

Not enough regs for all variables? We'll talk about that later…

Executing Add



Address	Instruction
1000	add $8, $4, $6
1004	add $5, $8, $7
1008	add $6, $6, $6
100C	…
1010	…

Memory

PC tells us where to execute next

Register	Value
$0	0000 0000
$1	1234 5678
$2	C001 D00D
$3	1BAD F00D
$4	9999 9999
$5	ABAB ABAB
$6	0000 0001
$7	0000 0002
$8	0000 0000
$9	0000 0000
$10	0000 0000
$11	0000 0000
$12	0000 0000
…	…
PC	**0000 1000**

RegFile

Executing Add

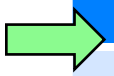
Address	Instruction
1000	add $8, $**4**, $**6**
1004	add $5, $8, $7
1008	add $6, $6, $6
100C	…
1010	…

Add reads its source registers, and uses their values directly (“register direct”)

9999 9999
0000 0001
9999 999A

Register	Value
$0	0000 0000
$1	1234 5678
$2	C001 D00D
$3	1BAD F00D
$4	**9999 9999**
$5	ABAB ABAB
$6	**0000 0001**
$7	0000 0002
$8	0000 0000
$9	0000 0000
$10	0000 0000
$11	0000 0000
$12	0000 0000
…	…
PC	0000 1000

Executing Add



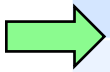
Address	Instruction
1000	add $**8**, $4, $6
1004	add $5, $8, $7
1008	add $6, $6, $6
100C	…
1010	…

Add writes its result to its destination register

Register	Value
$0	0000 0000
$1	1234 5678
$2	C001 D00D
$3	1BAD F00D
$4	9999 9999
$5	ABAB ABAB
$6	0000 0001
$7	0000 0002
$8	**9999 999A**
$9	0000 0000
$10	0000 0000
$11	0000 0000
$12	0000 0000
…	…
PC	0000 1000

Executing Add

Address	Instruction
1000	add $8, $4, $6
1004	add $5, $8, $7
1008	add $6, $6, $6
100C	…
1010	…



Register	Value
$0	0000 0000
$1	1234 5678
$2	C001 D00D
$3	1BAD F00D
$4	9999 9999
$5	ABAB ABAB
$6	0000 0001
$7	0000 0002
$8	9999 999A
$9	0000 0000
$10	0000 0000
$11	0000 0000
$12	0000 0000
…	…
PC	**0000 1004**

And goes to the sequentially next instruction
(PC = PC + 4)

Why 4? Counts in bytes (8 bits)
32-bit system → 32/8 = 4 bytes per insn

Executing Add

Address	Instruction
1000	add $8, $4, $6
1004	add $5, $8, $7
1008	add $6, $6, $6
100C	…
1010	…

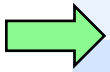
We set $5 equal to
$8 (9999 999A) + $7 (2) = 9999 999C

and PC = PC +4

Register	Value
$0	0000 0000
$1	1234 5678
$2	C001 D00D
$3	1BAD F00D
$4	9999 9999
$5	**9999 999C**
$6	0000 0001
$7	0000 0002
$8	9999 999A
$9	0000 0000
$10	0000 0000
$11	0000 0000
$12	0000 0000
…	…
PC	**0000 1008**

Executing Add

Address	Instruction
1000	add $8, $4, $6
1004	add $5, $8, $7
1008	add $6, $6, $6
100C	…
1010	…



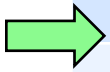
It's perfectly fine to have $6 as a src and a dst
This is just like x = x + x; in C, Java, etc:

1 + 1 = 2

Register	Value
$0	0000 0000
$1	1234 5678
$2	C001 D00D
$3	1BAD F00D
$4	9999 9999
$5	9999 999C
$6	0000 0001
$7	0000 0002
$8	9999 999A
$9	0000 0000
$10	0000 0000
$11	0000 0000
$12	0000 0000
…	…
PC	0000 1008

Executing Add

Address	Instruction
1000	add $8, $4, $6
1004	add $5, $8, $7
1008	add $6, $6, $6
100C	…
1010	…



Its perfectly fine to have $6 as a src and a dst
This is just like x = x + x; in C, Java, etc:

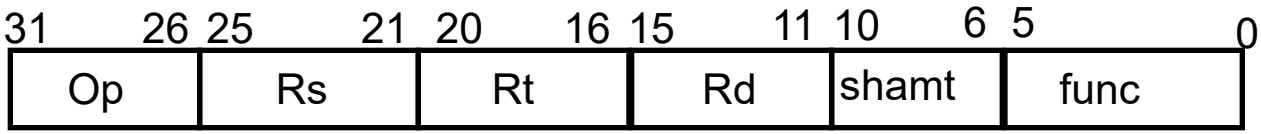
1 + 1 = 2

Register	Value
$0	0000 0000
$1	1234 5678
$2	C001 D00D
$3	1BAD F00D
$4	9999 9999
$5	9999 999C
$6	**0000 0002**
$7	0000 0002
$8	9999 999A
$9	0000 0000
$10	0000 0000
$11	0000 0000
$12	0000 0000
…	…
PC	**0000 100C**

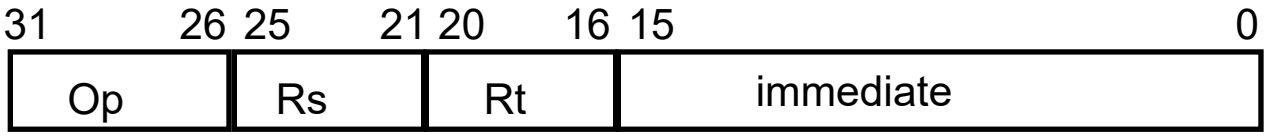
MIPS Instruction Formats

MIPS Instruction Formats

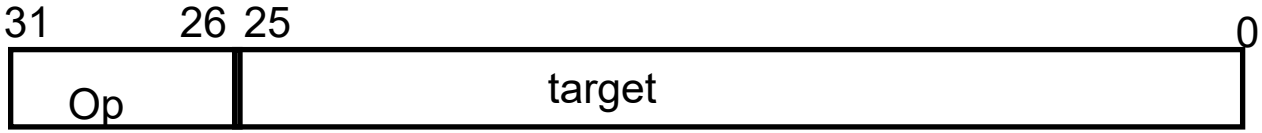
R-type: Register-Register



I-type: Register-Immediate



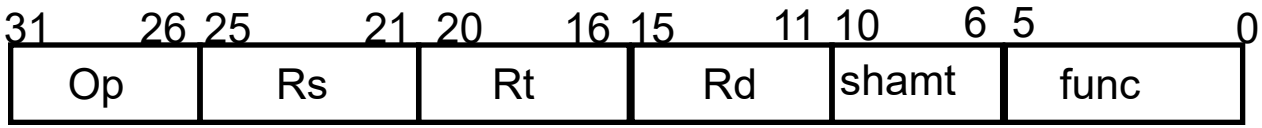
J-type: Jump / Call



Terminology
Op = opcode
Rs, Rt, Rd = register specifier

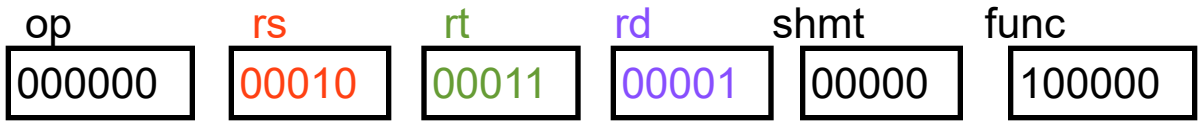
R Type: <OP> rd, rs, rt

R-type: Register-Register



op	a 6-bit operation code.
rs	a 5-bit source register.
rt	a 5-bit target (source) register.
rd	a 5-bit destination register.
shmt	a 5-bit shift amount.
func	a 6-bit function field (to identify more specific operations that have the same op).

Example: add $1, $2, $3



Executing add (with insns values)

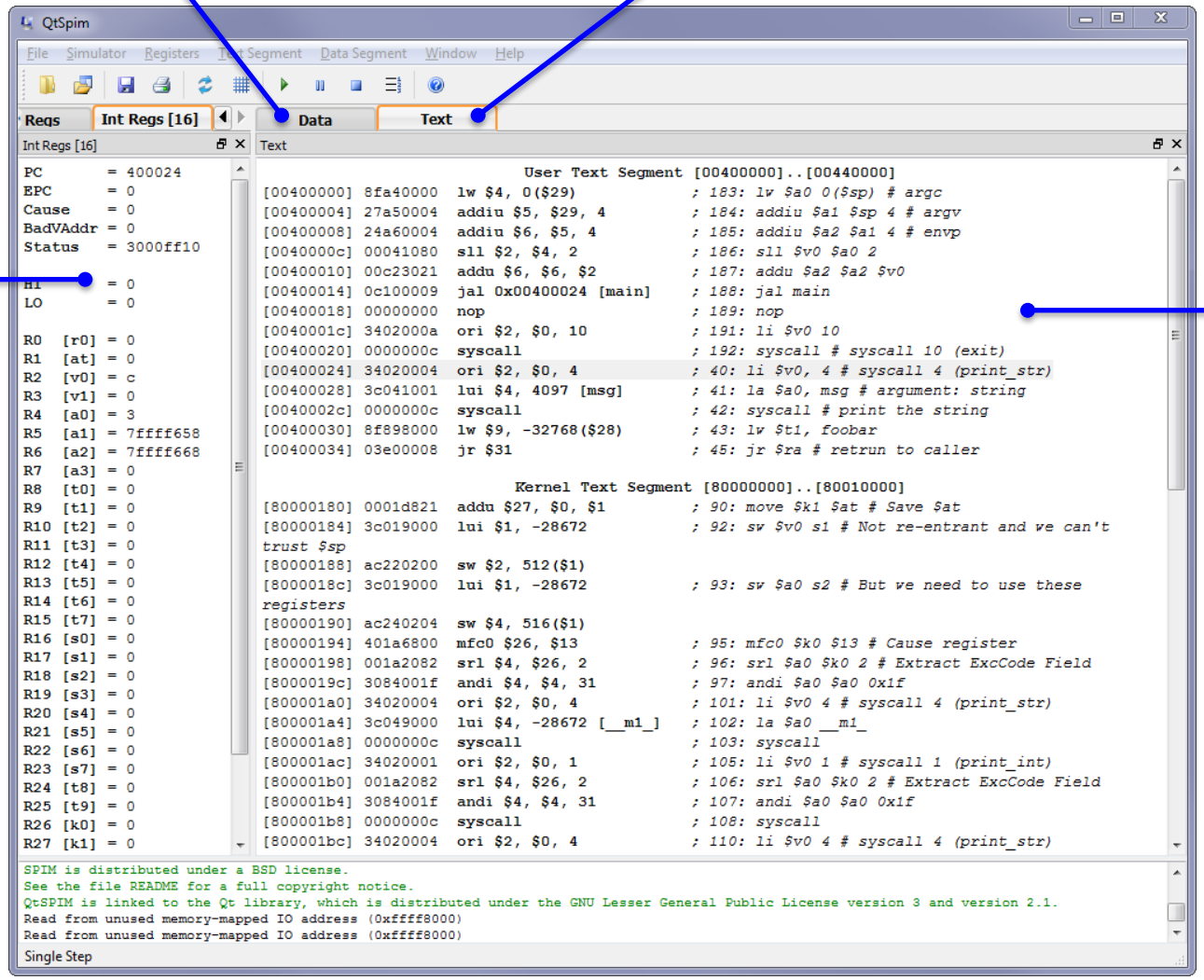
Address	Instruction (assembly)	Machine Code
1000	add $8, $4, $6	0086 4020
1004	add $5, $8, $7	0107 2820
1008	add $6, $6, $6	00C6 3020
100C	…	…
1010	…	…

Register	Value
$0	0000 0000
$1	1234 5678
$2	C001 D00D
$3	1BAD F00D
$4	9999 9999
$5	9999 999C
$6	0000 0002
$7	0000 0002
$8	9999 999A
$9	0000 0000
$10	0000 0000
…	…
PC	0000 100C

qtspim: Shows you similar state

Data memory is in this tab

Your program is in the “text” region of memory



Registers

Code

Other Similar Instructions

- `sub $rDest, $rSrc1, $rSrc2`
- `mul $rDest, $rSrc1, $rSrc2` (pseudo-insn)
- `div $rDest, $rSrc1, $rSrc2` (pseudo-insn)
- `and $rDest, $rSrc1, $rSrc2`
- `or  $rDest, $rSrc1, $rSrc2`
- `xor $rDest, $rSrc1, $rSrc2`
- …

Pseudo Instructions

- Some "instructions" are pseudo-instructions
  - Actually, assemble into 2 instructions:

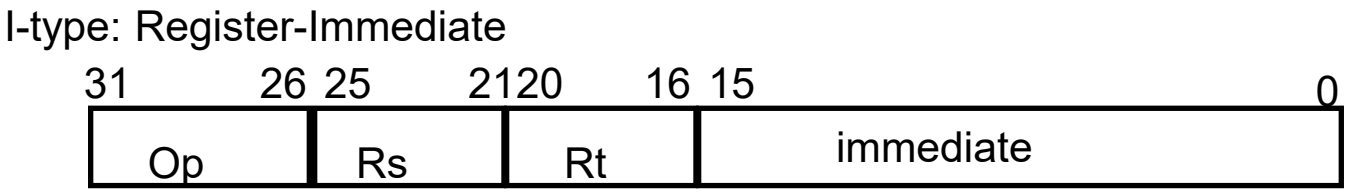
- `mul $1, $2, $3` is really

```
mul $2, $3
mflo $1
```

- mul takes two srcs, writes to special regs lo and hi
- mflo moves from lo into dst reg

I-Type <op> rt, rs, immediate

- Immediate: 16-bit value



Add Immediate Example
addi $1, $2, 100

Using addi to Put a Constant in a Register

- Can use addi to put a constant into a register:
  - x = 42;
  - Can be done with
  - addi $7, $0, 42
  - Because $0 is always 0.

- Common enough it has its own pseudo-insn:
  - `li $7, 42`
  - Stands for load immediate, works for 16-bit immediate

Many Insns Have Immediate Forms

- Add is not the only one with an immediate form
  - `andi`, `ori`, `xori`, `sll`, `sr`, `sra`, …
- No subi in MIPS
  - Why not?
    - Can always use addi with negation
- No muli or divi in MIPS
  - Though some ISAs have them

Assembly Programming Exercise

- Consider the following C fragment that converts temperature:

```
int tempF = 87;
int a = tempF - 32;
a = a * 5;
int tempC = a / 9;
```

- Let's write assembly for it
  - First, need registers for our variables:
    - `tempF = $3`
    - `a = $4`
    - `tempC = $5`
  - Now, give it a try (use $6, $7,… as temps if you need)…

Assembly Programming Exercise

- Consider the following C fragment that converts temperature:

```
int tempF = 87;
int a = tempF - 32;
a = a * 5;
int tempC = a / 9;
```

```
li $3, 87

addi $4, $3, -32

li $6, 5
mul $4, $4, $6

li $6, 9
div $5, $4, $6
```

- Let's write assembly for it
  - First, need registers for our variables:
    - `tempF = $3`
    - `a = $4`
    - `tempC = $5`
  - Now, give it a try (use $6, $7,… as temps if you need)…

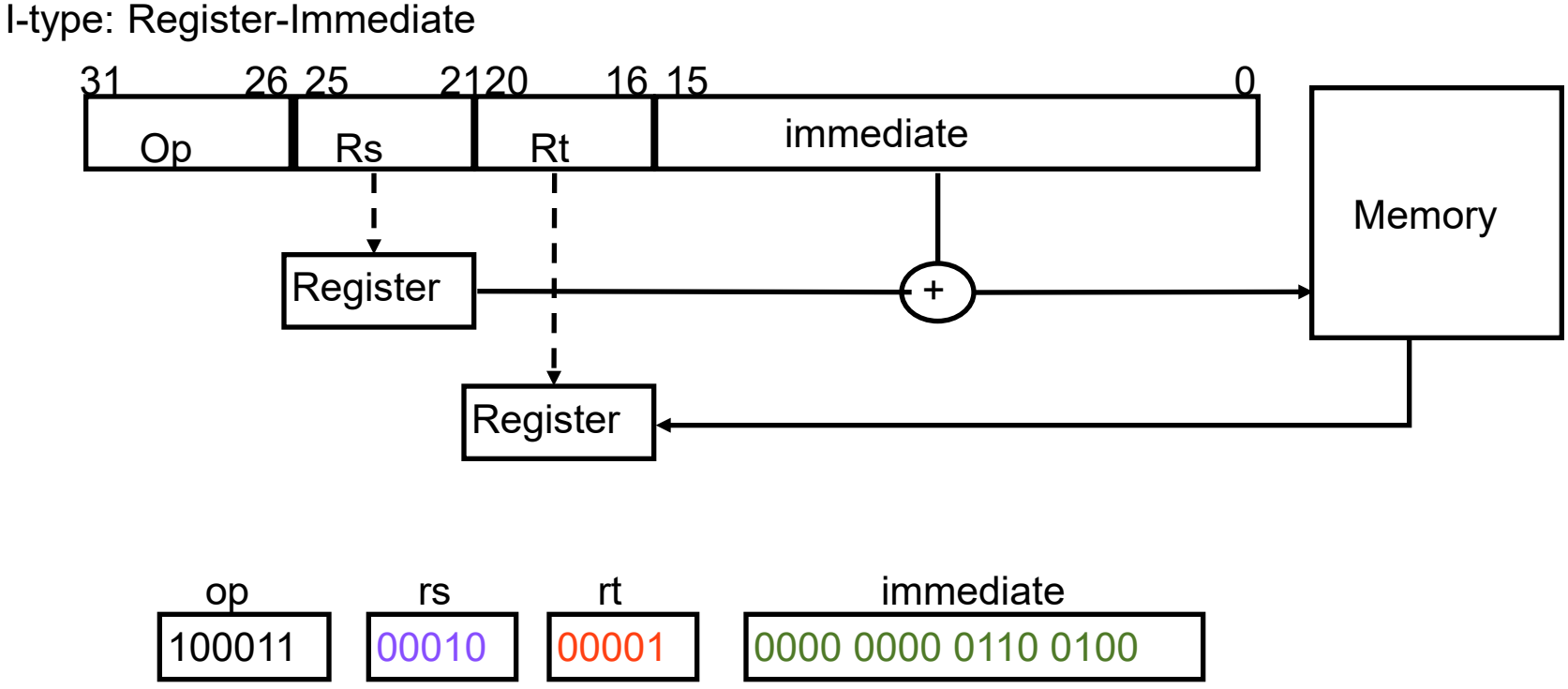
MIPS' ISA Is a "Load-Store" ISA

```
loop:   ~~lw $1, Memory[1004]~~ → lw $1, 0($6) # put val of x in $1
        ~~lw $2, Memory[1008]~~ → lw $2, 4($6) # put val of y in $2
        add $3, $1, $2
        add $4, $4, $3
        j loop
```

More on the meaning of the `lw` insn in the next slide

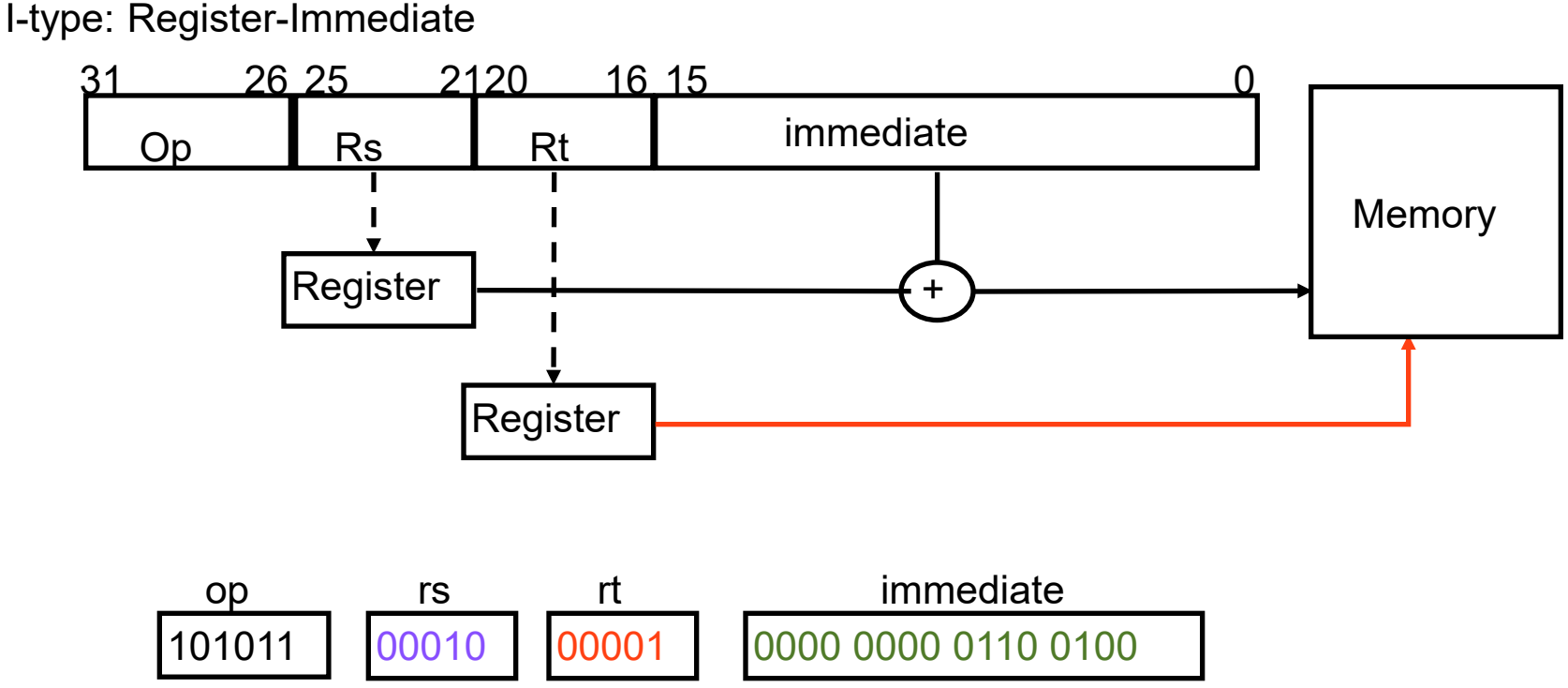
More I-Type Operations/Instructions

- Load Word Example
- `lw $1, 100($2)` # $1 = Mem[$2+100]



More I-Type Operations/Instructions

- Store Word Example
- `sw $1, 100($2)` # Mem[$2+100] = $1



Data Sizes/Types

- Loads and Stores come in multiple sizes
  - Reflect different data types
- The ‘w’ in lw/sw stands for “word” (= 32 bits)
  - Can also load bytes (8 bits) and half words (16 bits)
    - Smaller sizes have signed/unsigned forms

C to Assembly Exercise: Loads/Stores

- int x
- int * p
- int ** q
- …
- x = *p;
- **q = x;

- p = *q;
- p[4] = x;

```
# x in $1
# p in $2
# q in $3
…
```

C to Assembly Exercise: Loads/Stores

- int x
- int * p
- int ** q
- …
- x = *p;
- **q = x;

- p = *q;
- p[4] = x;

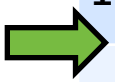
```
# x in $1
# p in $2
# q in $3
…
lw $1, 0($2)
lw $4, 0($3)
sw $1, 0($4)
lw $2, 0($3)
sw $1, 16($2)
```

Executing Memory Ops Example

Address	Value
1000	lw $1, 0($2)
1004	lw $4, 4($3)
1008	sw $1, 8($4)
100C	lw $2, 0($3)
1010	…
…	…
8000	F00D F00D
8004	C001 D00D
8008	1234 4321
800C	4242 4242
8010	0000 8000

Register	Value
$0	0000 0000
$1	1234 5678
$2	0000 8004
$3	0000 800C
$4	9999 9999
$5	9999 999C
$6	0000 0002
$7	0000 0002
$8	9999 999A
$9	0000 0000
$10	0000 0000
$11	0000 0000
$12	0000 0000
…	…
PC	0000 1000

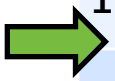
Executing Memory Ops Example



Address	Value
1000	lw $1, 0($2)
1004	lw $4, 4($3)
1008	sw $1, 8($4)
100C	lw $2, 0($3)
1010	…
…	…
8000	F00D F00D
8004	C001 D00D
8008	1234 4321
800C	4242 4242
8010	0000 8000

Register	Value
$0	0000 0000
$1	C001 D00D
$2	0000 8004
$3	0000 800C
$4	9999 9999
$5	9999 999C
$6	0000 0002
$7	0000 0002
$8	9999 999A
$9	0000 0000
$10	0000 0000
$11	0000 0000
$12	0000 0000
…	…
PC	0000 1004

Executing Memory Ops Example



Address	Value
1000	lw $1, 0($2)
1004	lw $4, 4($3)
1008	sw $1, 8($4)
100C	lw $2, 0($3)
1010	…
…	…
8000	F00D F00D
8004	C001 D00D
8008	1234 4321
800C	4242 4242
8010	0000 8000

Register	Value
$0	0000 0000
$1	C001 D00D
$2	0000 8004
$3	0000 800C
$4	0000 8000
$5	9999 999C
$6	0000 0002
$7	0000 0002
$8	9999 999A
$9	0000 0000
$10	0000 0000
$11	0000 0000
$12	0000 0000
…	…
PC	0000 1008

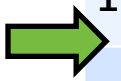
Executing Memory Ops Example

Address	Value
1000	lw $1, 0($2)
1004	lw $4, 4($3)
1008	sw $1, 8($4)
100C	lw $2, 0($3)
1010	…
…	…
8000	F00D F00D
8004	C001 D00D
8008	C001 D00D
800C	4242 4242
8010	0000 8000

Register	Value
$0	0000 0000
$1	C001 D00D
$2	0000 8004
$3	0000 800C
$4	0000 8000
$5	9999 999C
$6	0000 0002
$7	0000 0002
$8	9999 999A
$9	0000 0000
$10	0000 0000
$11	0000 0000
$12	0000 0000
…	…
PC	0000 100C

Executing Memory Ops Example

Address	Value
1000	lw $1, 0($2)
1004	lw $4, 4($3)
1008	sw $1, 8($4)
100C	lw $2, 0($3)
1010	…
…	…
8000	F00D F00D
8004	C001 D00D
8008	C001 D00D
800C	4242 4242
8010	0000 8000

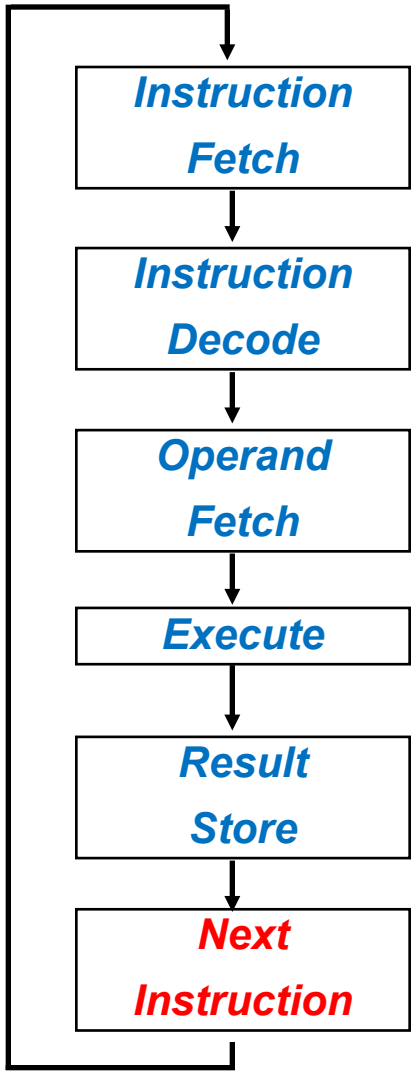


Register	Value
$0	0000 0000
$1	C001 D00D
$2	4242 4242
$3	0000 800C
$4	0000 8000
$5	9999 999C
$6	0000 0002
$7	0000 0002
$8	9999 999A
$9	0000 0000
$10	0000 0000
$11	0000 0000
$12	0000 0000
…	…
PC	0000 1010

Making Control Decisions

- Control constructs—decide what to do next:

```
if (x == y) {
…
}
else {
…
}
…
while (z < q) {
…
}
```



The Program Counter (PC)

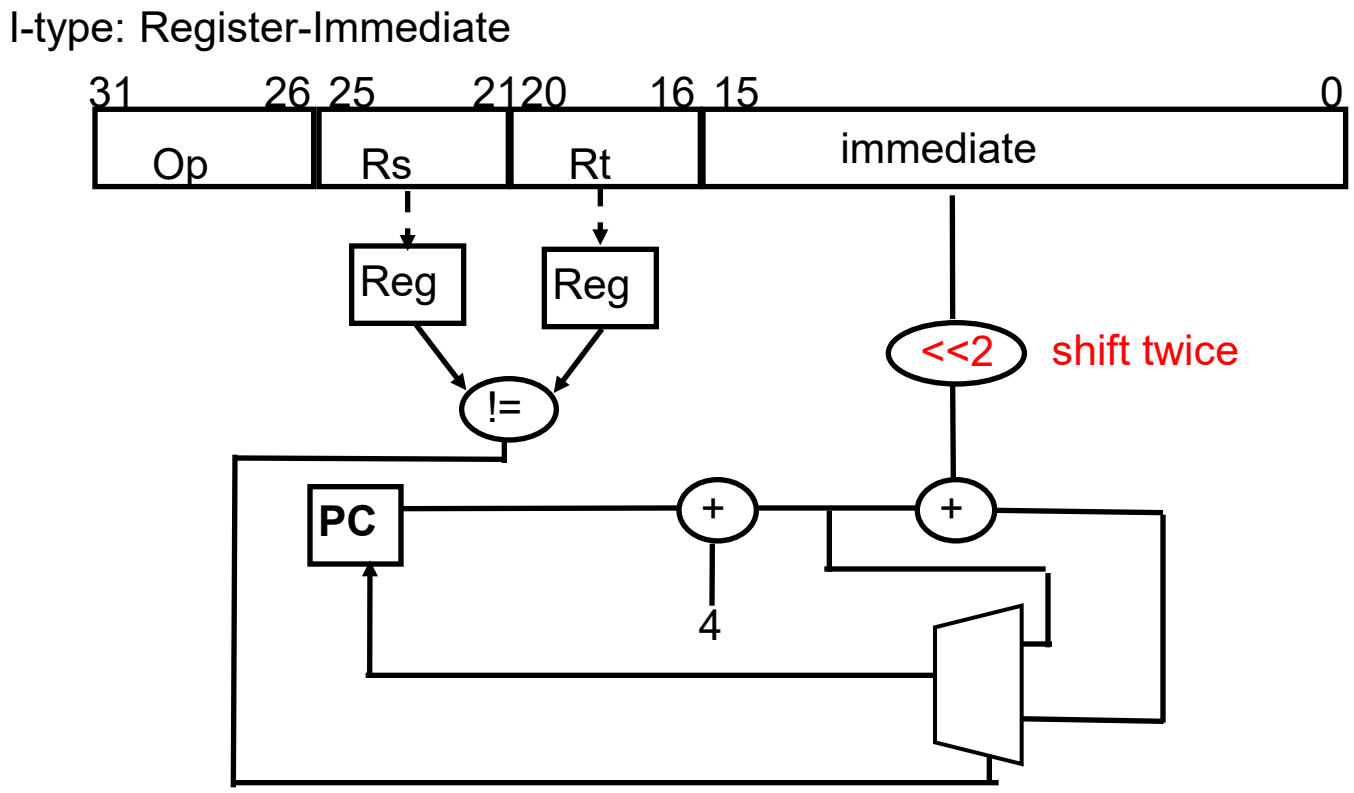
- Special register (PC) that points to instructions
- Contains memory address (like a pointer)

- During instruction fetch:
  - insn = mem[PC]

- So far, we have fetched sequentially: PC = PC + 4
  - Because memory is **byte-addressable** while MIPS insns are **32 bits** each
    Has unique address per byte
  - May want to specify non-sequential fetch
  - Change PC in other ways

I-Type <op> rt, rs, immediate

- PC relative addressing
- Branch if Not Equal Example:
  - bne $1, $2, 100 # If ($1!= $2) goto [PC+4+400]
    (not 100)



I-Type <op> rt, rs, immediate

- PC relative addressing
- Branch if Not Equal Example:
  - bne $1, $2, 100 # If ($1!= $2) goto [PC+4+400]
    (not 100)

- Why 400, not 100?
  - Examine the last 2 bits of PC
  - They're always "00"!
    - Since we always increment by 4
  - → So, we always save space when providing the immediate address by omitting the "00" at the end
    - But we need to bring the "00" back when computing PC!
    - → Shift immediate value twice left (i.e., value*4) to bring back "00"

MIPS Compare and Branch

- Compare and Branch
  - `beq  rs, rt, offset`
  - `bne  rs, rt, offset`
- Compare to zero and Branch
  - `blez  rs, offset`
  - `bgtz  rs, offset`
  - `bltz  rs, offset`
  - `bgez  rs, offset`
- Also pseudo-insns for unconditional branch (`b`)

- What do all these instructions mean/do??
  - Check the posted document "MIPS ISA.pdf" on Canvas
  - Also:
    - https://en.wikipedia.org/wiki/MIPS_architecture
    - https://en.wikibooks.org/wiki/MIPS_Assembly/Instruction_Formats

MIPS Jump, Branch, Compare Instructions

- Inequality to something other than 0: requires 2 insns
  - Conditionally set reg, branch if not zero or if zero

- `slt $1,$2,$3` $1 = ($2<$3) ? 1 : 0
  - Compare less than; signed 2's comp.

- `slti $1,$2,100` $1 = ($2 < 100) ? 1 : 0
  - Compare < constant; signed 2's comp.

- `sltu $1,$2,$3` $1 = ($2 < $3) ? 1 : 0
  - Compare less than; unsigned

- `sltiu $1,$2,100` $1 = ($2 < 100) ? 1 : 0 $1=0
  - Compare < constant; unsigned

- `beqz $1,100` if ($1 == 0) go to PC+4+400
  - Branch if equal to 0

- `bnez $1,100` if ($1!= 0) go to PC+4+400
  - Branch if not equal to 0

Signed vs. Unsigned Comparison

- $1= 0…00 0000 0000 0000 0001
- $2= 0…00 0000 0000 0000 0010
- $3= 1…11 1111 1111 1111 1111
- After executing these instructions:

```
slt  $4,$2,$1
slt  $5,$3,$1
sltu $6,$2,$1
sltu $7,$3,$1
```

- What are values of registers $4 - $7?

  $4 = 0 ; $5 = 1 ; $6 = 0 ; $7 = 0

C to Assembly with Branches Exercise

```
int x;                          //assume x in $1
int y;                          //assume y in $2
int z;                          //assume z in $3
…                               …
if (x != y) {
   z = z + 2;
}
else {
   z = z - 4;
}
```

C to Assembly with Branches Exercise

```
int x;                                  //assume x in $1
int y;                                  //assume y in $2
int z;                                  //assume z in $3
…                                       …
if (x != y) {                           beq $1, $2, L_else
   z = z + 2;                           addi $3, $3, 2
}                                       b L_end
else {
   z = z - 4;                L_else:    addi $3, $3, -4
}                            L_end:
```

J-Type <op> immediate

- 16-bit imm limits us to +/- 32K insns
- Usually fine, but sometimes we need more…
- J-type insns provide long range, unconditional jump:

- Specifies lowest 28 bits of PC (26 bits followed by "00")
  - Upper 4 bits of PC are unchanged
  - Range: 64 Million instruction (256 MB)

- Can also jump anywhere using `jr $reg#` (jump register)

Calling Functions

Back to the Fahrenheit to Celsius Program…

- Consider the following C fragment:

```
int tempF = 87;
int a = tempF – 32;
a = a * 5;

int tempC = a / 9;
```

```
li   $3, 87
addi $4, $3, -32
li   $6, 5
mul  $4, $4, $6
li   $6, 9
div  $5, $4, $6
```

- If we were really doing this…

  We would abstract it into a **function** and call it

More Likely: A Function

- Like this:

```
int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
}
…
…
int tempC = f2c(87);
```

We Need a Way to Call f2c and Return

- Call: Jump… but also remember where to go back
  - There may be many calls to f2c() in the program
  - We need some way to know where to return
  - Instruction for this **jal**
    - `jal label`
    - Store PC+4 into $31
    - Jump to label (memory address/pointer)

- Return: Jump… back to wherever we were
  - Instruction for this **jr**
    - `jr $31`
  - Jump back to address stored by jal in $31

f2c

- Like this:

```
int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
}                                   //jr $31
…
…
int tempC = f2c(87);                //jal f2c
```

- But that's not all…

f2c

- Like this:

```
int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
}                                   //jr $31
…
…
int tempC = f2c(87);                //jal f2c
```

- Still need to pass 87 as argument to f2c

f2c

- Like this:

```
int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
}                                   //jr $31
…
…
int tempC = f2c(87);                //jal f2c
```

- Need to return `tempC`

f2c

- Like this:

```
int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
}                                   //jr $31
…
…
int tempC = f2c(87);                //jal f2c
```

- Also, may want to reuse the same registers later
  - What if f2c called something? That would re-use $31

Calling Convention

- All of these are reasons for a calling convention
  - Agreement of how registers are used
  - Where arguments are passed, results returned
  - Who must save what if they want to use it
  - Etc.

MIPS Register Usage/Naming Conventions

0	zero	constant
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary (caller saves)
. . .		
15	t7	

16	s0	saved for later (callee saves)
. . .		
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return address

Also 32 floating-point registers: $f0 .. $f31

Important: The only general-purpose registers are the $s and $t registers. Everything else has a specific usage:

$a = arguments, $v = return values, $ra = return address, etc.

‘t’ vs. ‘s’ Registers

- Caller saves (t) registers
  - If some code is about to call another function…
  - And it needs the value in a caller saves register ($t0,$t1…)
  - Then it has to save it on the **stack** before the call
  - And restore it after the call

- Callee saves (s) registers
  - If some code wants to use a callee saves register (at all)
  - It has to save it to the stack before it uses it
  - And restore it before it returns to its caller
  - But, it can assume any function it calls will not change the register
    - Either won’t use it, or will save/restore it

- →
  - **t: temporary variable for the current function**
  - **s: variable to be saved for later use (after other function calls)**

f2c

```
int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
}
...
...
int tempC = f2c(87)
```

Try to convert this into assembly given the conventions we just learned

f2c

```
int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
}
```

```
f2c:
    addi $t0, $a0, -32
```

tempF is in $a0 by calling convention

f2c

```
int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
}
```

```
f2c:
    addi $t0, $a0, -32
```

We can use $t0 for a temp (like a) without saving it

f2c

```
int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
}
```

```
f2c:
    addi $t0, $a0, -32
    li $t1, 5
    mul $t0, $t0, $t1
    li $t1, 9
    div $t2, $t0, $t1
```

f2c

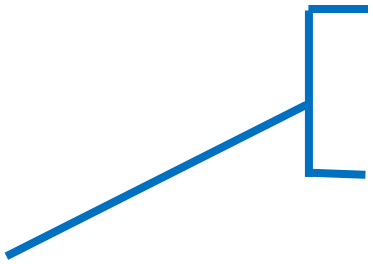
```
int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
}
```

```
f2c:
    addi $t0, $a0, -32
    li $t1, 5
    mul $t0, $t0, $t1
    li $t1, 9
    div $t2, $t0, $t1
    addi $v0, $t2, 0
    jr $ra
```

f2c

```
int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
}
```

```
f2c:
    addi $t0, $a0, -32
    li $t1, 5
    mul $t0, $t0, $t1
    li $t1, 9
    div $t2, $t0, $t1
    addi $v0, $t2, 0
    jr $ra
```



A smart compiler would just do
div $v0, $t0, $t1

f2c

```
int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
}
...
...
int tempC = f2c(87)
```

```
f2c:
    addi $t0, $a0, -32
    li $t1, 5
    mul $t0, $t0, $t1
    li $t1, 9
    div $t2, $t0, $t1
    addi $v0, $t2, 0
    jr $ra
…
addi $a0, $0, 87
```

f2c

```
int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
}
...
...
int tempC = f2c(87)
```

```
f2c:
    addi $t0, $a0, -32
    li $t1, 5
    mul $t0, $t0, $t1
    li $t1, 9
    div $t2, $t0, $t1
    addi $v0, $t2, 0
    jr $ra
…
addi $a0, $0, 87
jal f2c
```

f2c

```
int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
}
...
...
int tempC = f2c(87)
```

```
f2c:
    addi $t0, $a0, -32
    li $t1, 5
    mul $t0, $t0, $t1
    li $t1, 9
    div $t2, $t0, $t1
    addi $v0, $t2, 0
    jr $ra
…
addi $a0, $0, 87
jal f2c
addi $t0, $v0, 0
```

What It Would Take To Make SPIM Happy

```
.globl f2c     # f2c can be called from any file
.text          # goes in "text" region

f2c:           #   (remember memory picture?)
addi $t0, $a0, -32
li $t1, 5
mul $t0, $t0, $t1
li $t1, 9
div $t2, $t0, $t1
addi $v0, $t2, 0
jr $ra

.end f2c       # end of this function
```

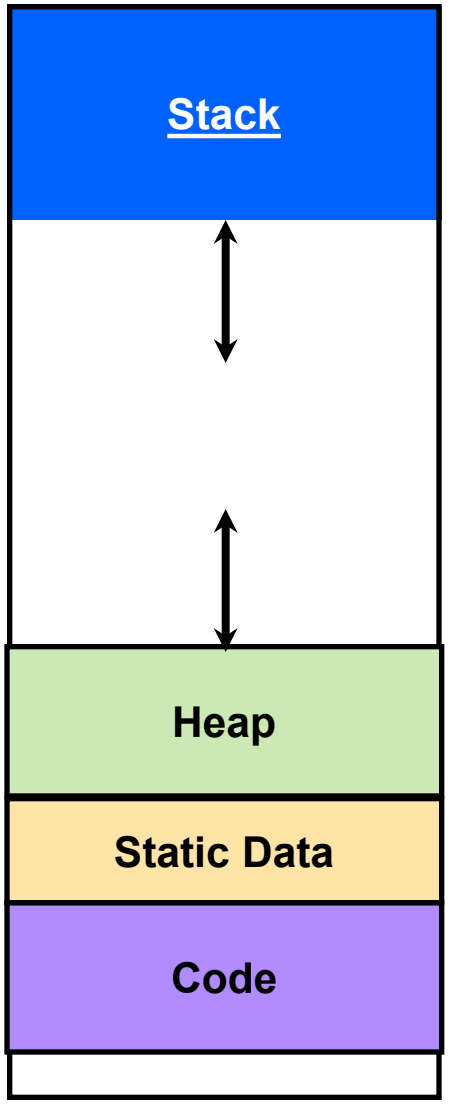
SPIM Assembly Language Sugar

- Directives: tell the assembler what to do...
- Format "."<string> [arg1], [arg2] …

- Examples
  - .data # start a data segment.
  - .text # start a code segment.
  - .align n # align segment on 2^n byte boundary.
  - .ascii <string> # store a string in memory.
  - .asciiz <string> # store a null terminated string in memory
  - .word w1, w2, . . . , wn # store n words in memory.
  - .space n # reserve n bytes of space

Function Calls? → Let's Check Out The Stack

- May need to use the stack for…
  - Saving registers
    - Across calls
    - Spilling variables (not enough registers)
  - Passing more than 4 arguments
  - Other variables…



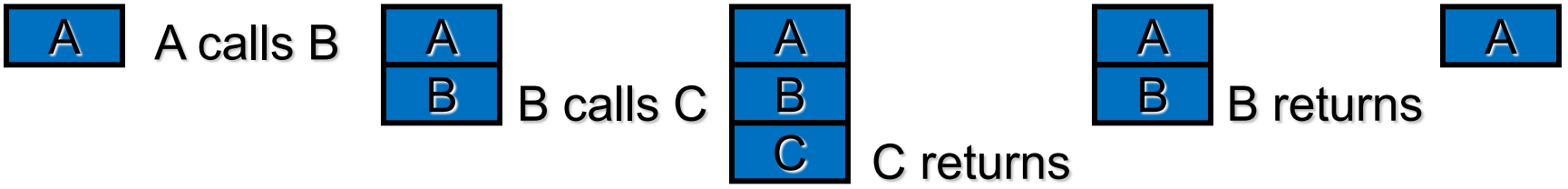
Stack Layout

- Stack is in the memory of every executing process:
  - Loads and stores are used to access it
  - But what address to load/store?

- Two registers for referencing stack:
  - Stack pointer ($sp): Points to end (bottom) of stack
  - Frame pointer ($fp): Points to top of current stack frame

Procedures Use the Stack

- In general, procedure calls obey stack discipline
  - Local procedure state contained in stack frame
  - Where we can save registers
  - When a procedure is called, a new frame opens (frame is "pushed")
  - When a procedure returns, the frame collapses (frame is "popped")
- Procedure stack is in memory
  - Starts is at the "top" of the memory and grows down



MIPS/GCC Procedure Calling Conventions

Calling procedure (before jumping to called function):

- Step-1: Setup the arguments:
  - The first four arguments (arg0-arg3) are passed in registers $a0-$a3
  - Remaining arguments are pushed onto the stack
    (in reverse order, arg5 is at the top of the stack)
- Step-2: Save needed temporary registers
  - Save $t regs to $s regs if they contain values needed after function call
- Step-3: Execute jal
- Step-4: Clean up stack (if more than 4 args)

MIPS/GCC Procedure Calling Conventions

Called routine (before executing function's body):

- Step-1: Establish stack frame
  - Subtract the frame size from the stack pointer
  - Typically, minimum frame size is 32 bytes (8 words)

- Step-2: Save needed regs in the frame
  - $fp is always saved
  - $ra is saved if routine makes a call
  - $s0-$s7 are saved if they are used

- Step-3: Establish frame pointer
  - Add the stack <frame size> - 4 to the address in $sp
  - *Note that frame pointer isn't strictly necessary, but it helps with debugging (when using gdb, for example)*

MIPS/GCC Procedure Calling Conventions

On return from a call (after executing function's body):

- Step-1: Put returned values in registers $v0, [$v1]
  (if values are returned)

- Step-2: Restore registers
  - Restore $fp and other saved registers [$ra, $s0 - $s7]

- Step-3: Pop the stack
  - Add the frame size to $sp

- Step-4: Return
  - Jump to the address in $ra

```
jr      $ra
```

Execution Example: Calling With Frames

Addr	Instruction
1000	addi $sp, $sp, -16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addi $fp, $sp, 12
1014	add $s0, $a0, $a1
1018	jal 4200
101C	add $t0, $v0, $s0
1020	lw $v0, 4($t0)
1024	lw $s0, -4($fp)
1028	lw $ra, -8($fp)
102C	lw $fp, -12($fp)
1030	addi $sp, $sp, 16
1034	jr $ra

Just did jal 1000

Reg	Value
$0	0000 0000
$at	0000 0000
$v0	4242 4242
$v1	0000 8010
$a0	0000 1234
$a1	5678 0001
$a2	0000 0002
$a3	0000 0007
$t0	9999 999A
$t1	0000 0000
$s0	0042 0420
$sp	0000 FFE0
$fp	0000 FFF0
$ra	0000 2348
PC	0000 1000

Addr	Value
FFF0	0001 0070
FFEC	1234 5678
FFE8	9999 9999
FFE4	0000 2568
FFE0	0001 0040
FFDC	
FFD8	
FFD4	
FFD0	
FFCC	
FFC8	
FFC4	
FFC0	
FFBC	

Execution Example: Calling With Frames

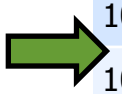
Addr	Instruction
1000	addi $sp, $sp, -16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addi $fp, $sp, 12
1014	add $s0, $a0, $a1
1018	jal 4200
101C	add $t0, $v0, $s0
1020	lw $v0, 4($t0)
1024	lw $s0, -4($fp)
1028	lw $ra, -8($fp)
102C	lw $fp, -12($fp)
1030	addi $sp, $sp, 16
1034	jr $ra

$sp, $fp still describe caller's frame

Reg	Value
$0	0000 0000
$at	0000 0000
$v0	4242 4242
$v1	0000 8010
$a0	0000 1234
$a1	5678 0001
$a2	0000 0002
$a3	0000 0007
$t0	9999 999A
$t1	0000 0000
$s0	0042 0420
$sp	0000 FFE0
$fp	0000 FFF0
$ra	0000 2348
PC	0000 1000

Addr	Value
FFF0	0001 0070
FFEC	1234 5678
FFE8	9999 9999
FFE4	0000 2568
FFE0	0001 0040
FFDC	
FFD8	
FFD4	
FFD0	
FFCC	
FFC8	
FFC4	
FFC0	
FFBC	

Execution Example: Calling With Frames



Addr	Instruction
1000	addi $sp, $sp, -16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addi $fp, $sp, 12
1014	add $s0, $a0, $a1
1018	jal 4200
101C	add $t0, $v0, $s0
1020	lw $v0, 4($t0)
1024	lw $s0, -4($fp)
1028	lw $ra, -8($fp)
102C	lw $fp, -12($fp)
1030	addi $sp, $sp, 16
1034	jr $ra

Allocate space on stack

Reg	Value
$0	0000 0000
$at	0000 0000
$v0	4242 4242
$v1	0000 8010
$a0	0000 1234
$a1	5678 0001
$a2	0000 0002
$a3	0000 0007
$t0	9999 999A
$t1	0000 0000
$s0	0042 0420
$sp	**0000 FFD0**
$fp	0000 FFF0
$ra	0000 2348
PC	0000 1004

Addr	Value
FFF0	0001 0070
FFEC	1234 5678
FFE8	9999 9999
FFE4	0000 2568
FFE0	0001 0040
FFDC	
FFD8	
FFD4	
FFD0	
FFCC	
FFC8	
FFC4	
FFC0	
FFBC	

Execution Example: Calling With Frames

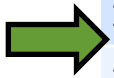
Addr	Instruction
1000	addi $sp, $sp, -16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addi $fp, $sp, 12
1014	add $s0, $a0, $a1
1018	jal 4200
101C	add $t0, $v0, $s0
1020	lw $v0, 4($t0)
1024	lw $s0, -4($fp)
1028	lw $ra, -8($fp)
102C	lw $fp, -12($fp)
1030	addi $sp, $sp, 16
1034	jr $ra

Save $fp

Reg	Value
$0	0000 0000
$at	0000 0000
$v0	4242 4242
$v1	0000 8010
$a0	0000 1234
$a1	5678 0001
$a2	0000 0002
$a3	0000 0007
$t0	9999 999A
$t1	0000 0000
$s0	0042 0420
$sp	0000 FFD0
$fp	0000 FFF0
$ra	0000 2348
PC	0000 1008

Addr	Value
FFF0	0001 0070
FFEC	1234 5678
FFE8	9999 9999
FFE4	0000 2568
FFE0	0001 0040
FFDC	
FFD8	
FFD4	
FFD0	**0000 FFF0**
FFCC	
FFC8	
FFC4	
FFC0	
FFBC	

Execution Example: Calling With Frames



Addr	Instruction
1000	addi $sp, $sp, -16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addi $fp, $sp, 12
1014	add $s0, $a0, $a1
1018	jal 4200
101C	add $t0, $v0, $s0
1020	lw $v0, 4($t0)
1024	lw $s0, -4($fp)
1028	lw $ra, -8($fp)
102C	lw $fp, -12($fp)
1030	addi $sp, $sp, 16
1034	jr $ra

Save $ra

Reg	Value
$0	0000 0000
$at	0000 0000
$v0	4242 4242
$v1	0000 8010
$a0	0000 1234
$a1	5678 0001
$a2	0000 0002
$a3	0000 0007
$t0	9999 999A
$t1	0000 0000
$s0	0042 0420
$sp	0000 FFD0
$fp	0000 FFF0
$ra	0000 2348
PC	0000 100C

Addr	Value
FFF0	0001 0070
FFEC	1234 5678
FFE8	9999 9999
FFE4	0000 2568
FFE0	0001 0040
FFDC	
FFD8	
FFD4	**0000 2348**
FFD0	0000 FFF0
FFCC	
FFC8	
FFC4	
FFC0	
FFBC	

Execution Example: Calling With Frames

Addr	Instruction
1000	addi $sp, $sp, -16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addi $fp, $sp, 12
1014	add $s0, $a0, $a1
1018	jal 4200
101C	add $t0, $v0, $s0
1020	lw $v0, 4($t0)
1024	lw $s0, -4($fp)
1028	lw $ra, -8($fp)
102C	lw $fp, -12($fp)
1030	addi $sp, $sp, 16
1034	jr $ra

Save $s0

Reg	Value
$0	0000 0000
$at	0000 0000
$v0	4242 4242
$v1	0000 8010
$a0	0000 1234
$a1	5678 0001
$a2	0000 0002
$a3	0000 0007
$t0	9999 999A
$t1	0000 0000
$s0	0042 0420
$sp	0000 FFD0
$fp	0000 FFF0
$ra	0000 2348
PC	0000 1010

Addr	Value
FFF0	0001 0070
FFEC	1234 5678
FFE8	9999 9999
FFE4	0000 2568
FFE0	0001 0040
FFDC	
FFD8	**0042 0420**
FFD4	0000 2348
FFD0	0000 FFF0
FFCC	
FFC8	
FFC4	
FFC0	
FFBC	

Execution Example: Calling With Frames

Addr	Instruction
1000	addi $sp, $sp, -16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addi $fp, $sp, 12
1014	add $s0, $a0, $a1
1018	jal 4200
101C	add $t0, $v0, $s0
1020	lw $v0, 4($t0)
1024	lw $s0, -4($fp)
1028	lw $ra, -8($fp)
102C	lw $fp, -12($fp)
1030	addi $sp, $sp, 16
1034	jr $ra

Setup $fp

Reg	Value
$0	0000 0000
$at	0000 0000
$v0	4242 4242
$v1	0000 8010
$a0	0000 1234
$a1	5678 0001
$a2	0000 0002
$a3	0000 0007
$t0	9999 999A
$t1	0000 0000
$s0	0042 0420
$sp	0000 FFD0
$fp	**0000 FFDC**
$ra	0000 2348
PC	0000 1014

Addr	Value
FFF0	0001 0070
FFEC	1234 5678
FFE8	9999 9999
FFE4	0000 2568
FFE0	0001 0040
FFDC	
FFD8	0042 0420
FFD4	0000 2348
FFD0	0000 FFF0
FFCC	
FFC8	
FFC4	
FFC0	
FFBC	

Execution Example: Calling With Frames

Addr	Instruction
1000	addi $sp, $sp, -16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addi $fp, $sp, 12
1014	add $s0, $a0, $a1
1018	jal 4200
101C	add $t0, $v0, $s0
1020	lw $v0, 4($t0)
1024	lw $s0, -4($fp)
1028	lw $ra, -8($fp)
102C	lw $fp, -12($fp)
1030	addi $sp, $sp, 16
1034	jr $ra

$sp, $fp now describe new frame, ready to start

Reg	Value
$0	0000 0000
$at	0000 0000
$v0	4242 4242
$v1	0000 8010
$a0	0000 1234
$a1	5678 0001
$a2	0000 0002
$a3	0000 0007
$t0	9999 999A
$t1	0000 0000
$s0	0042 0420
$sp	0000 FFD0
$fp	0000 FFDC
$ra	0000 2348
PC	0000 1014

Addr	Value
FFF0	0001 0070
FFEC	1234 5678
FFE8	9999 9999
FFE4	0000 2568
FFE0	0001 0040
FFDC	
FFD8	0042 0420
FFD4	0000 2348
FFD0	0000 FFF0
FFCC	
FFC8	
FFC4	
FFC0	
FFBC	

Execution Example: Calling With Frames

Addr	Instruction
1000	addi $sp, $sp, -16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addi $fp, $sp, 12
1014	add $s0, $a0, $a1
1018	jal 4200
101C	add $t0, $v0, $s0
1020	lw $v0, 4($t0)
1024	lw $s0, -4($fp)
1028	lw $ra, -8($fp)
102C	lw $fp, -12($fp)
1030	addi $sp, $sp, 16
1034	jr $ra

Do some computation..

Reg	Value
$0	0000 0000
$at	0000 0000
$v0	4242 4242
$v1	0000 8010
$a0	0000 1234
$a1	5678 0001
$a2	0000 0002
$a3	0000 0007
$t0	9999 999A
$t1	0000 0000
$s0	**5678 1235**
$sp	0000 FFD0
$fp	0000 FFDC
$ra	0000 2348
PC	0000 1018

Addr	Value
FFF0	0001 0070
FFEC	1234 5678
FFE8	9999 9999
FFE4	0000 2568
FFE0	0001 0040
FFDC	
FFD8	0042 0420
FFD4	0000 2348
FFD0	0000 FFF0
FFCC	
FFC8	
FFC4	
FFC0	
FFBC	

Execution Example: Calling With Frames

Addr	Instruction
1000	addi $sp, $sp, -16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addi $fp, $sp, 12
1014	add $s0, $a0, $a1
1018	jal 4200
101C	add $t0, $v0, $s0
1020	lw $v0, 4($t0)
1024	lw $s0, -4($fp)
1028	lw $ra, -8($fp)
102C	lw $fp, -12($fp)
1030	addi $sp, $sp, 16
1034	jr $ra

Call another function
(not pictured, takes no args)

Reg	Value
$0	0000 0000
$at	0000 0000
$v0	4242 4242
$v1	0000 8010
$a0	0000 1234
$a1	5678 0001
$a2	0000 0002
$a3	0000 0007
$t0	9999 999A
$t1	0000 0000
$s0	5678 1235
$sp	0000 FFD0
$fp	0000 FFDC
$ra	**0000 101C**
PC	0000 4200

Addr	Value
FFF0	0001 0070
FFEC	1234 5678
FFE8	9999 9999
FFE4	0000 2568
FFE0	0001 0040
FFDC	
FFD8	0042 0420
FFD4	0000 2348
FFD0	0000 FFF0
FFCC	
FFC8	
FFC4	
FFC0	
FFBC	

jal sets $ra, PC

Execution Example: Calling With Frames

Addr	Instruction
1000	addi $sp, $sp, -16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addi $fp, $sp, 12
1014	add $s0, $a0, $a1
1018	jal 4200
101C	add $t0, $v0, $s0
1020	lw $v0, 4($t0)
1024	lw $s0, -4($fp)
1028	lw $ra, -8($fp)
102C	lw $fp, -12($fp)
1030	addi $sp, $sp, 16
1034	jr $ra

Reg	Value
$0	0000 0000
$at	???? ????
$v0	???? ????
$v1	???? ????
$a0	???? ????
$a1	???? ????
$a2	???? ????
$a3	???? ????
$t0	???? ????
$t1	???? ????
$s0	???? ????
$sp	???? ????
$fp	???? ????
$ra	???? ????
PC	???? ????

Addr	Value
FFF0	0001 0070
FFEC	1234 5678
FFE8	9999 9999
FFE4	0000 2568
FFE0	0001 0040
FFDC	
FFD8	0042 0420
FFD4	0000 2348
FFD0	0000 FFF0
FFCC	
FFC8	
FFC4	
FFC0	
FFBC	

Other function can do what
It wants to the regs as it computes

And make a stack frame
Of its own

Execution Example: Calling With Frames

Addr	Instruction
1000	addi $sp, $sp, -16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addi $fp, $sp, 12
1014	add $s0, $a0, $a1
1018	jal 4200
101C	add $t0, $v0, $s0
1020	lw $v0, 4($t0)
1024	lw $s0, -4($fp)
1028	lw $ra, -8($fp)
102C	lw $fp, -12($fp)
1030	addi $sp, $sp, 16
1034	jr $ra

Reg	Value
$0	0000 0000
$at	???? ????
$v0	**8675 3090**
$v1	???? ????
$a0	???? ????
$a1	???? ????
$a2	???? ????
$a3	???? ????
$t0	???? ????
$t1	???? ????
$s0	**5678 1235**
$sp	**0000 FFD0**
$fp	**0000 FFDC**
$ra	**0000 101C**
PC	**0000 101C**

Addr	Value
FFF0	0001 0070
FFEC	1234 5678
FFE8	9999 9999
FFE4	0000 2568
FFE0	0001 0040
FFDC	
FFD8	0042 0420
FFD4	0000 2348
FFD0	0000 FFF0
FFCC	
FFC8	
FFC4	
FFC0	
FFBC	

But before it returns, it is responsible for restoring certain registers

Including $sp and $fp, and $s0
Value returned in $v0

Execution Example: Calling With Frames

Addr	Instruction
1000	addi $sp, $sp, -16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addi $fp, $sp, 12
1014	add $s0, $a0, $a1
1018	jal 4200
101C	add $t0, $v0, $s0
1020	lw $v0, 4($t0)
1024	lw $s0, -4($fp)
1028	lw $ra, -8($fp)
102C	lw $fp, -12($fp)
1030	addi $sp, $sp, 16
1034	jr $ra

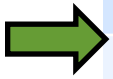
Do some more computation

Reg	Value
$0	0000 0000
$at	???? ????
$v0	8675 3090
$v1	???? ????
$a0	???? ????
$a1	???? ????
$a2	???? ????
$a3	???? ????
$t0	**DCED 42C5**
$t1	???? ????
$s0	5678 1235
$sp	0000 FFD0
$fp	0000 FFDC
$ra	0000 101C
PC	0000 1020

Addr	Value
FFF0	0001 0070
FFEC	1234 5678
FFE8	9999 9999
FFE4	0000 2568
FFE0	0001 0040
FFDC	
FFD8	0042 0420
FFD4	0000 2348
FFD0	0000 FFF0
FFCC	
FFC8	
FFC4	
FFC0	
FFBC	

Execution Example: Calling With Frames

Addr	Instruction
1000	addi $sp, $sp, -16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addi $fp, $sp, 12
1014	add $s0, $a0, $a1
1018	jal 4200
101C	add $t0, $v0, $s0
1020	lw $v0, 4($t0)
1024	lw $s0, -4($fp)
1028	lw $ra, -8($fp)
102C	lw $fp, -12($fp)
1030	addi $sp, $sp, 16
1034	jr $ra



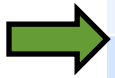
Do some more computation
(load addr not pictured)

Reg	Value
$0	0000 0000
$at	???? ????
$v0	**0001 0002**
$v1	???? ????
$a0	???? ????
$a1	???? ????
$a2	???? ????
$a3	???? ????
$t0	DCED 42C5
$t1	???? ????
$s0	5678 1235
$sp	0000 FFD0
$fp	0000 FFDC
$ra	0000 101C
PC	0000 1024

Addr	Value
FFF0	0001 0070
FFEC	1234 5678
FFE8	9999 9999
FFE4	0000 2568
FFE0	0001 0040
FFDC	
FFD8	0042 0420
FFD4	0000 2348
FFD0	0000 FFF0
FFCC	
FFC8	
FFC4	
FFC0	
FFBC	

Execution Example: Calling With Frames

Addr	Instruction
1000	addi $sp, $sp, -16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addi $fp, $sp, 12
1014	add $s0, $a0, $a1
1018	jal 4200
101C	add $t0, $v0, $s0
1020	lw $v0, 4($t0)
1024	lw $s0, -4($fp)
1028	lw $ra, -8($fp)
102C	lw $fp, -12($fp)
1030	addi $sp, $sp, 16
1034	jr $ra



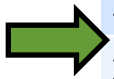
Restore registers to return

Reg	Value
$0	0000 0000
$at	???? ????
$v0	0001 0002
$v1	???? ????
$a0	???? ????
$a1	???? ????
$a2	???? ????
$a3	???? ????
$t0	DCED 42C5
$t1	???? ????
$s0	**0042 0420**
$sp	0000 FFD0
$fp	0000 FFDC
$ra	0000 101C
PC	0000 1028

Addr	Value
FFF0	0001 0070
FFEC	1234 5678
FFE8	9999 9999
FFE4	0000 2568
FFE0	0001 0040
FFDC	
FFD8	0042 0420
FFD4	0000 2348
FFD0	0000 FFF0
FFCC	
FFC8	
FFC4	
FFC0	
FFBC	

Execution Example: Calling With Frames

Addr	Instruction
1000	addi $sp, $sp, -16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addi $fp, $sp, 12
1014	add $s0, $a0, $a1
1018	jal 4200
101C	add $t0, $v0, $s0
1020	lw $v0, 4($t0)
1024	lw $s0, -4($fp)
1028	lw $ra, -8($fp)
102C	lw $fp, -12($fp)
1030	addi $sp, $sp, 16
1034	jr $ra



Restore registers to return

Reg	Value
$0	0000 0000
$at	???? ????
$v0	0001 0002
$v1	???? ????
$a0	???? ????
$a1	???? ????
$a2	???? ????
$a3	???? ????
$t0	DCED 42C5
$t1	???? ????
$s0	0042 0420
$sp	0000 FFD0
$fp	0000 FFDC
$ra	**0000 2348**
PC	0000 102C

Addr	Value
FFF0	0001 0070
FFEC	1234 5678
FFE8	9999 9999
FFE4	0000 2568
FFE0	0001 0040
FFDC	
FFD8	0042 0420
FFD4	0000 2348
FFD0	0000 FFF0
FFCC	
FFC8	
FFC4	
FFC0	
FFBC	

Execution Example: Calling With Frames

Addr	Instruction
1000	addi $sp, $sp, -16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addi $fp, $sp, 12
1014	add $s0, $a0, $a1
1018	jal 4200
101C	add $t0, $v0, $s0
1020	lw $v0, 4($t0)
1024	lw $s0, -4($fp)
1028	lw $ra, -8($fp)
102C	lw $fp, -12($fp)
1030	addi $sp, $sp, 16
1034	jr $ra

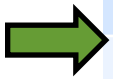
Restore registers to return

Reg	Value
$0	0000 0000
$at	???? ????
$v0	0001 0002
$v1	???? ????
$a0	???? ????
$a1	???? ????
$a2	???? ????
$a3	???? ????
$t0	DCED 42C5
$t1	???? ????
$s0	0042 0420
$sp	0000 FFD0
$fp	**0000 FFF0**
$ra	0000 2348
PC	0000 1030

Addr	Value
FFF0	0001 0070
FFEC	1234 5678
FFE8	9999 9999
FFE4	0000 2568
FFE0	0001 0040
FFDC	
FFD8	0042 0420
FFD4	0000 2348
FFD0	0000 FFF0
FFCC	
FFC8	
FFC4	
FFC0	
FFBC	

Execution Example: Calling With Frames

Addr	Instruction
1000	addi $sp, $sp, -16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addi $fp, $sp, 12
1014	add $s0, $a0, $a1
1018	jal 4200
101C	add $t0, $v0, $s0
1020	lw $v0, 4($t0)
1024	lw $s0, -4($fp)
1028	lw $ra, -8($fp)
102C	lw $fp, -12($fp)
1030	addi $sp, $sp, 16
1034	jr $ra



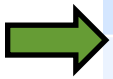
Restore registers to return

Reg	Value
$0	0000 0000
$at	???? ????
$v0	0001 0002
$v1	???? ????
$a0	???? ????
$a1	???? ????
$a2	???? ????
$a3	???? ????
$t0	DCED 42C5
$t1	???? ????
$s0	0042 0420
$sp	**0000 FFE0**
$fp	0000 FFF0
$ra	0000 2348
PC	0000 1034

Addr	Value
FFF0	0001 0070
FFEC	1234 5678
FFE8	9999 9999
FFE4	0000 2568
FFE0	0001 0040
FFDC	
FFD8	0042 0420
FFD4	0000 2348
FFD0	0000 FFF0
FFCC	
FFC8	
FFC4	
FFC0	
FFBC	

Execution Example: Calling With Frames

Addr	Instruction
1000	addi $sp, $sp, -16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addi $fp, $sp, 12
1014	add $s0, $a0, $a1
1018	jal 4200
101C	add $t0, $v0, $s0
1020	lw $v0, 4($t0)
1024	lw $s0, -4($fp)
1028	lw $ra, -8($fp)
102C	lw $fp, -12($fp)
1030	addi $sp, $sp, 16
1034	jr $ra



Restore registers to return

Reg	Value
$0	0000 0000
$at	???? ????
$v0	0001 0002
$v1	???? ????
$a0	???? ????
$a1	???? ????
$a2	???? ????
$a3	???? ????
$t0	DCED 42C5
$t1	???? ????
$s0	0042 0420
$sp	0000 FFE0
$fp	0000 FFF0
$ra	0000 2348
PC	0000 1034

Addr	Value
FFF0	0001 0070
FFEC	1234 5678
FFE8	9999 9999
FFE4	0000 2568
FFE0	0001 0040
FFDC	
FFD8	0042 0420
FFD4	0000 2348
FFD0	0000 FFF0
FFCC	
FFC8	
FFC4	
FFC0	
FFBC	

Now $sp, $fp describe caller's frame

Execution Example: Calling With Frames

Addr	Instruction
1000	addi $sp, $sp, -16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addi $fp, $sp, 12
1014	add $s0, $a0, $a1
1018	jal 4200
101C	add $t0, $v0, $s0
1020	lw $v0, 4($t0)
1024	lw $s0, -4($fp)
1028	lw $ra, -8($fp)
102C	lw $fp, -12($fp)
1030	addi $sp, $sp, 16
1034	jr $ra

Return to caller
(code not pictured)

Reg	Value
$0	0000 0000
$at	???? ????
$v0	0001 0002
$v1	???? ????
$a0	???? ????
$a1	???? ????
$a2	???? ????
$a3	???? ????
$t0	DCED 42C5
$t1	???? ????
$s0	0042 0420
$sp	0000 FFE0
$fp	0000 FFF0
$ra	0000 2348
PC	0000 2348

Addr	Value
FFF0	0001 0070
FFEC	1234 5678
FFE8	9999 9999
FFE4	0000 2568
FFE0	0001 0040
FFDC	
FFD8	0042 0420
FFD4	0000 2348
FFD0	0000 FFF0
FFCC	
FFC8	
FFC4	
FFC0	
FFBC	

Assembly Writing Tips and Advice

- Write C first (steps!), then translate C → Assembly
  - One function at a time
  - Pick registers for each variable
    - Must be in memory? Give it a stack slot (refer to by $fp+num)
    - Must live across a procedure call? Use an $s register
      - Otherwise, use a $t
  - Write prolog (beginning)
    - Save ra/fp (if needed)
    - Save any $s registers you use
  - Translate line by line
  - Write epilog (end)
    - Kind of the opposite actions of the prolog
    - Store returns (if needed)
    - Then jr

Why Do We Need FP?

- The frame pointer is not always required
  - Can often get away without it
- When/why do we need it?
  - Debugging tools (like gdb) use it to find frames
  - If you have variable length arrays
    - Stack pointer changes by amount not known at compile time
    - But variables would still be at constant offset from frame pointer
- How to reference stuff without it?
  - Everything would be offset from the stack pointer: 4($sp), 8($sp), etc.
- Good practice for this class to use it
  - Don't prematurely optimize

System Call Instruction

- System call is used to communicate with the operating system and request services (memory allocation, I/O)
  - syscall instruction in MIPS
  - Sort of like a procedure call, but call to ask OS for help

- SPIM supports "system calls lite"

1. Load system call code into register $v0
   - Example: if $v0==1, then syscall will print an integer
2. Load arguments (if any) into registers $a0, $a1, or $f12 (for floating point)
3. `syscall`
4. Results returned in registers $v0 or $f0

SPIM System Call Support

code	service	ArgType	Arg/Result
1	print	int	$a0
2	print	float	$f12
3	print	double	$f12
4	print	string	$a0 (string address)
5	read	integer	integer in $v0
6	read	float	float in $f0
7	read	double	double in $f0 & $f1
8	read	string	$a0=buffer, $a1=length
9	sbrk	$a0=amount	address in $v0
10	exit		

Plus a few more for general file IO which we shouldn't need.

Echo Number and String

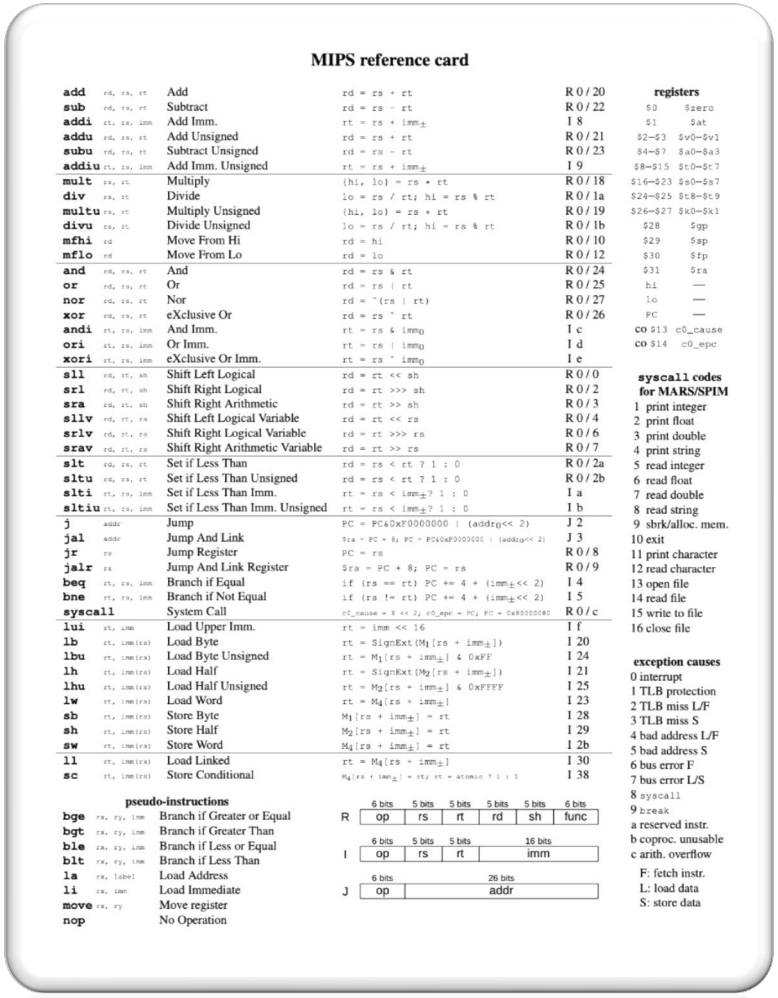
```
.text
main:
    li      $v0, 5              # code to read an integer
    syscall                     # do the read (invokes the OS)
    move     $a0, $v0           # copy result from $v0 to $a0

    li      $v0, 1              # code to print an integer
    syscall                     # print the integer

    li      $v0, 4              # code to print string
    la      $a0, nln            # address of string (newline)
    syscall

    li      $v0, 8              # code to read a string
    la      $a0, name           # address of buffer (name)
    li      $a1, 8              # size of buffer (8 bytes)
    syscall

    la      $a0, name           # address of string to print
    li      $v0, 4              # code to print a string
    syscall

    jr      $31                 # return

    .data
    .align 2
name:       .word 0,0
nln:        .asciiz "\n"
```

More MIPS Assembly Rules

- One instruction per line.
- Numbers are base-10 integers or Hex w/ leading 0x.
- Identifiers: alphanumeric, '_', '.' string starting in a letter or _
- Labels: identifiers starting at the beginning of a line followed by ":"
- Comments: everything following # till end-of-line.
- Instruction format: Space and "," separated fields.
  - [Label:] <op> reg1, [reg2], [reg3] [# comment]
  - [Label:] <op> reg1, offset(reg2) [# comment]
  - .Directive [arg1], [arg2], . . .

MIPS in One Page

- Check the quick reference posted on Canvas under Files → Reference Documents



MIPS reference card

add	rd, rs, rt	Add	rd = rs + rt	R 0 / 20
sub	rd, rs, rt	Subtract	rd = rs - rt	R 0 / 22
addi	rt, rs, imm	Add Imm.	rt = rs + imm±	I 8
addu	rd, rs, rt	Add Unsigned	rd = rs + rt	R 0 / 21
subu	rd, rs, rt	Subtract Unsigned	rd = rs - rt	R 0 / 23
addiu	rt, rs, imm	Add Imm. Unsigned	rt = rs + imm±	I 9
mult	rs, rt	Multiply	{hi, lo} = rs * rt	R 0 / 18
div	rs, rt	Divide	lo = rs / rt; hi = rs % rt	R 0 / 1a
multu	rs, rt	Multiply Unsigned	{hi, lo} = rs * rt	R 0 / 19
divu	rs, rt	Divide Unsigned	lo = rs / rt; hi = rs % rt	R 0 / 1b
mfhi	rd	Move From Hi	rd = hi	R 0 / 10
mflo	rd	Move From Lo	rd = lo	R 0 / 12
and	rd, rs, rt	And	rd = rs & rt	R 0 / 24
or	rd, rs, rt	Or	rd = rs | rt	R 0 / 25
nor	rd, rs, rt	Nor	rd = ~(rs | rt)	R 0 / 27
xor	rd, rs, rt	eXclusive Or	rd = rs ^ rt	R 0 / 26
andi	rt, rs, imm	And Imm.	rt = rs & imm0	I c
ori	rt, rs, imm	Or Imm.	rt = rs | imm0	I d
xori	rt, rs, imm	eXclusive Or Imm.	rt = rs ^ imm0	I e
sll	rd, rt, sh	Shift Left Logical	rd = rt << sh	R 0 / 0
srl	rd, rt, sh	Shift Right Logical	rd = rt >>> sh	R 0 / 2
sra	rd, rt, sh	Shift Right Arithmetic	rd = rt >> sh	R 0 / 3
sllv	rd, rt, rs	Shift Left Logical Variable	rd = rt << rs	R 0 / 4
srlv	rd, rt, rs	Shift Right Logical Variable	rd = rt >>> rs	R 0 / 6
srav	rd, rt, rs	Shift Right Arithmetic Variable	rd = rt >> rs	R 0 / 7
slt	rd, rs, rt	Set if Less Than	rd = rs < rt ? 1 : 0	R 0 / 2a
sltu	rd, rs, rt	Set if Less Than Unsigned	rd = rs < rt ? 1 : 0	R 0 / 2b
slti	rt, rs, imm	Set if Less Than Imm.	rt = rs < imm±? 1 : 0	I a
sltiu	rt, rs, imm	Set if Less Than Imm. Unsigned	rt = rs < imm±? 1 : 0	I b
j	addr	Jump	PC = PC&0xF0000000 | (addr<< 2)	J 2
jal	addr	Jump And Link	$ra = PC + 8; PC = PC&0xF0000000 | (addr<< 2)	J 3
jr	rs	Jump Register	PC = rs	R 0 / 8
jalr	rs	Jump And Link Register	$ra = PC + 8; PC = rs	R 0 / 9
beq	rt, rs, imm	Branch if Equal	if (rs == rt) PC += 4 + (imm±<< 2)	I 4
bne	rt, rs, imm	Branch if Not Equal	if (rs != rt) PC += 4 + (imm±<< 2)	I 5
syscall		System Call	c0_cause = 8 << 2; c0_epc = PC; PC = 0x80000080	R 0 / c
lui	rt, imm	Load Upper Imm.	rt = imm << 16	I f
lb	rt, imm(rs)	Load Byte	rt = SignExt(M1[rs + imm±])	I 20
lbu	rt, imm(rs)	Load Byte Unsigned	rt = M1[rs + imm±] & 0xFF	I 24
lh	rt, imm(rs)	Load Half	rt = SignExt(M2[rs + imm±])	I 21
lhu	rt, imm(rs)	Load Half Unsigned	rt = M2[rs + imm±] & 0xFFFF	I 25
lw	rt, imm(rs)	Load Word	rt = M4[rs + imm±]	I 23
sb	rt, imm(rs)	Store Byte	M1[rs + imm±] = rt	I 28
sh	rt, imm(rs)	Store Half	M2[rs + imm±] = rt	I 29
sw	rt, imm(rs)	Store Word	M4[rs + imm±] = rt	I 2b
ll	rt, imm(rs)	Load Linked	rt = M4[rs + imm±]	I 30
sc	rt, imm(rs)	Store Conditional	M4[rs + imm±] = rt; rt = atomic ? 1 : 0	I 38

pseudo-instructions

bge	rx, ry, lbl	Branch if Greater or Equal
bgt	rx, ry, lbl	Branch if Greater Than
ble	rx, ry, lbl	Branch if Less or Equal
blt	rx, ry, lbl	Branch if Less Than
la	rx, label	Load Address
li	rx, imm	Load Immediate
move	rx, ry	Move register
nop		No Operation

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R	op	rs	rt	rd	sh	func
	6 bits	5 bits	5 bits	16 bits		
I	op	rs	rt	imm		
	6 bits	26 bits				
J	op	addr				

registers

$0	$zero
$1	$at
$2–$3	$v0–$v1
$4–$7	$a0–$a3
$8–$15	$t0–$t7
$16–$23	$s0–$s7
$24–$25	$t8–$t9
$26–$27	$k0–$k1
$28	$gp
$29	$sp
$30	$fp
$31	$ra
hi	—
lo	—
PC	—
co $13	c0_cause
co $14	c0_epc

syscall codes for MARS/SPIM

1 print integer
2 print float
3 print double
4 print string
5 read integer
6 read float
7 read double
8 read string
9 sbrk/alloc. mem.
10 exit
11 print character
12 read character
13 open file
14 read file
15 write to file
16 close file

exception causes

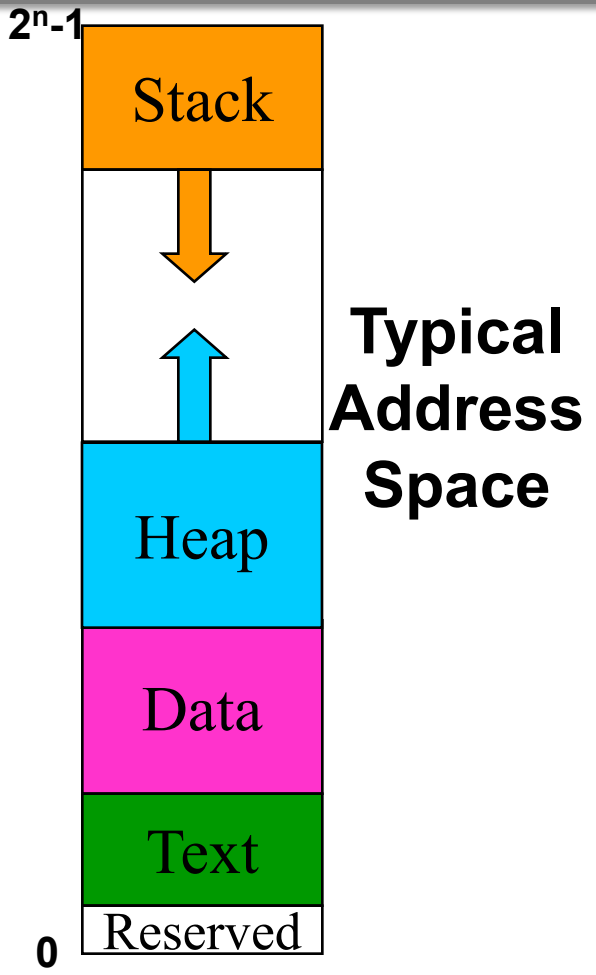
0 interrupt
1 TLB protection
2 TLB miss L/F
3 TLB miss S
4 bad address L/F
5 bad address S
6 bus error F
7 bus error L/S
8 syscall
9 break
a reserved instr.
b coproc. unusable
c arith. overflow

F: fetch instr.
L: load data
S: store data

Mapping C Variables to Memory

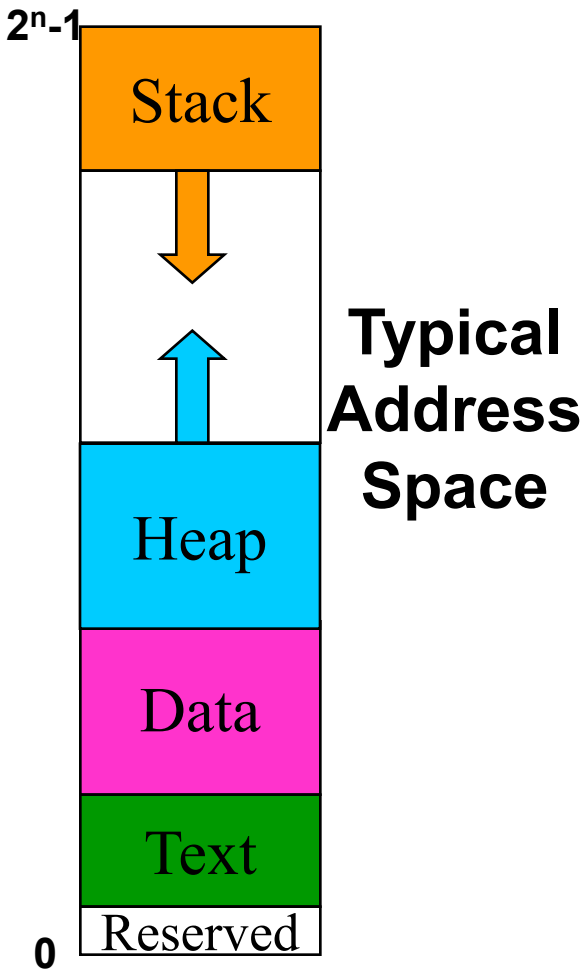
Memory Layout

- Memory is array of bytes, but there are conventions as to what goes where in this array
- Text: instructions (the program to execute)
- Data: global variables
- Stack: local variables and other per-function state; starts at top & grows downward
- Heap: dynamically allocated variables; grows upward
- What if stack and heap overlap????



Memory Layout: Example

```
int anumber = 3;

int factorial (int x) {
  if (x == 0) {
    return 1;
  }
  else {
    return x * factorial (x – 1);
  }
}

int main (void) {
  int z = factorial (anumber);
  printf("%d\n", z);
  return 0;
}
```



Memory Layout: Example

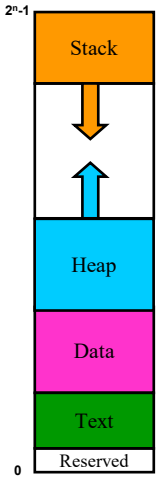
Global
```
int anumber = 3;
```

Param
```
int factorial (int x) {
  if (x == 0) {
    return 1;
  }
  else {
    return x * factorial (x - 1);
  }
}
```

```
int main (void) {
```

Local
```
  int z = factorial (anumber);
  printf("%d\n", z);
  return 0;
}
```

```
.data
anumber: .word 3

.text
.globl factorial
factorial:
; find x in $a0.
; if more args than 4,
; find them in stack
  ...

.globl main
main:
; make local z exist by
; subtracting 4 from $sp
addiu $sp, $sp, -4
  ...
```



What Is an Array?

- Array is nothing but a contiguous chunk of memory.
- The name of the array is a pointer to the beginning of the chunk of memory array has.
- So int array[100] is a contiguous chunk of memory which is of size 100*4 = 400 bytes.
- so array is almost of type int*
  (not exactly int* to be very exact but good enough for our purposes)

Adapted from “Arrays and Pointers” by Anand George, SourceLens.org. Link.

So...

- C pointer notation vs. array notation:
  - array[0] is equal to *(array + 0) or *(array)
  - array[1] is equal to *(array + 1)
  - array[2] is equal to *(array + 2)
  - …

- In C pointer math, the units are *elements*
- In MIPS memory accesses, the units are *bytes*

- If `array` lives at 0x1000, then:
  - `array[0]` is `*(array+0)`, and it lives at 0x1000+0*4 = 0x1000
  - `array[1]` is `*(array+1)`, and it lives at 0x1000+1*4 = 0x1004
  - `array[2]` is `*(array+2)`, and it lives at 0x1000+2*4 = 0x1008
  - …

Adapted from “Arrays and Pointers” by Anand George, SourceLens.org. Link.

Pointers vs. Arrays

- How they're similar:
  - Both are represented by a memory address which has one or more bytes of content at it

- How they differ:
  - Pointers store a memory address in memory. Only allocated one word (4 bytes on MIPS)
    - Global:

```
char* name=NULL;  →   .data
                      name: .word 0
```

    - Local:

```
char* name=NULL;  →   addiu $sp, $sp, -16 ; assuming 3 other words needed
                      sw $0, 12($sp)
```

  - Arrays allocated enough space for the array itself. Array declaration by itself doesn't store a memory *address* into memory, compiler just knows where the array lives and makes references to it use that memory region.
    - Global:

```
char name[40]="";  →  .data
                      name: .space 40 ; allocates 40 bytes, all null
```

    - Local:

```
char name[40]="";  →  addiu $sp, $sp, -52 ; assuming 3 other words needed
                      sb $0, 12($sp) ; only need to set first byte to null
```

What About String Literals?

- Pointers to a string literal: put string in read-only data region, store address to it.
  - Global:

```
char* name="hello"; →      .data
                           name: .word str_hello
                           .rdata
                           str_hello: .asciiz "hello"
```

  - Local:

```
char* name="hello"; →      addiu $sp, $sp, -16 ; assuming 3 other words needed
                           la $t0, str_hello
                           sw $t0, 12($sp)
```

`la` is "Load address": a pseudo-isn to put an address into a register.

- Arrays set to string literal: Allocate space for the string and initialize it.
  - Global:

```
char name[40]="hello"; →   .data
                           name: .asciiz "hello"
                           .space 34
```

  - Local:

```
char name[40]="hello"; →   addiu $sp, $sp, -52 ; assuming 3 other words needed
                           add $a0, $sp, 12 ; destination for copy
                           la $a1, str_hello ; source for copy
                           jal memset ; std function to copy memory
```

Pointers vs. Arrays

- When you *pass* an array to a function or store a *reference* to an array, then you're using pointers.

```
int func(int* ar) {...}

int my_array[50];
func(my_array);
```

- `ar` gets the address of the start of `my_array`
- Still true for this syntax:

```
int func(int ar[]) {...}
int func(int ar[40]) {...}
```

Multidimensional Arrays

- Again, contiguous chunk of data.
- Behaves like
  - array of arrays in the case of 2 dimensions.
  - array of array of array in 3 dimension and so on.

- All are different interpretations and syntaxes on single chunk of contiguous memory

Adapted from “Arrays and Pointers” by Anand George, SourceLens.org. Link.

Multidimensional Arrays Syntax

```
int mytable[10][20];
```

- We have 10 arrays of array of 20 integers.
- So total 10*20*4 = 800 bytes of contiguous memory.
- As before, the array name acts as a memory address to the beginning of the array.
- Type of the pointer is sort of like `int**`, but the row size needs to be known, so it's actually `int (*mytable)[20]`

- **How to access?** Pointer arithmetic.

```
myarray[x][y]
```
is equal to
```
*(myarray+x*20+y)
```


Adapted from "Arrays and Pointers" by Anand George, SourceLens.org. Link.

2D arrays in MIPS

- Global:

```
int a[10][20];
int b[10][20] = {{1,2,3,...,20},{101,102,...,120},...}
```

→

```
.data
a: .space 800
b: .word 1,2,3,…,20,101,...
```

- Local:

```
int a[10][20];
int b[10][20] = {{1,2,3,...,20},{101,102,...,120},...}
```

→

```
addiu $sp, $sp, -1612 ; assuming 3 other words needed
; no initialization for a, so it has whatever trash was on the stack already
li $t0,1
sw $t0, 12($sp)
li $t0,2
sw $t0, 16($sp)
li $t0,3
sw $t0, 20($sp)
  ... ; lots of initialization code
```

What about `sizeof`?

- **`sizeof`** tells you how big something is, in bytes

- For variables declared as **arrays**, this is the size of the array:

```
int array[10]; // sizeof(array) is 10*4 = 40 bytes
int array2d[10][20] // sizeof(array2d) is 10*20*4=800
```

- For pointers, this is the size of one pointer:

```
int* p; // sizeof(p) is 4 bytes
        // (assuming we're on a 32-bit system)
```

- This is true even if the pointer is used to refer to an array:

```
p = array; // p gets the address of the array.
           // sizeof(p) is still 4 bytes
```