ECE 550: Fundamentals of Computer Systems and Engineering

Instruction Set Architecture

MIPS

Admin

- Homework #2 Out now
  - Due October 2nd
  - VHDL part: more work than hwk1
- Reading
  - Chapter 2

Last time…

- Who can remind us what we did last time?

Last time…

- Who can remind us what we did last time?
  - Finite State Machines
  - Division

Now: Moving to software

- C is pretty low level
  - Most of you: in 551, can program in C
    - Others: assume you know programming, pointers, etc.
  - But compiled into **assembly**…
- Lower level programming
  - What does assembly look like?
  - How does software communicate with hardware?
  - What variations can there be?
  - Advantages/disadvantages?

Assembly

- Assembly programming:
  - 1 (sometimes two) **machine instructions** at a time
  - Still in "human readable form"
    - add r1, r2, r3
  - Much "lower level" than any other programming
    - Limited number of **registers** vs unlimited variables
    - Flat scope
      - (who can remind us what scope is? Hint: not mouthwash)

Registers

- Two places processors can store data
  - Registers (saw these---sort of):
    - In processor
    - Few of them (e.g., 32)
    - Fast (more on this much later in semester)
  - Memory (later):
    - Outside of processor
    - Huge (e.g., 4GB)
    - Slow (generally about 100—200x slower than registers, more later)
- For now: think of registers like "variables"
  - But only 32 of them
  - E.g., r1 = r2 + r3 much like x = y + z

Assembly too high level for machine

- Human readable is not (easily) machine executable
  - add r1, r2, r3
- **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.)
- **Assembler** does this translation
  - Humans don't typically need to write raw bits

What Must be Specified?

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

The ISA

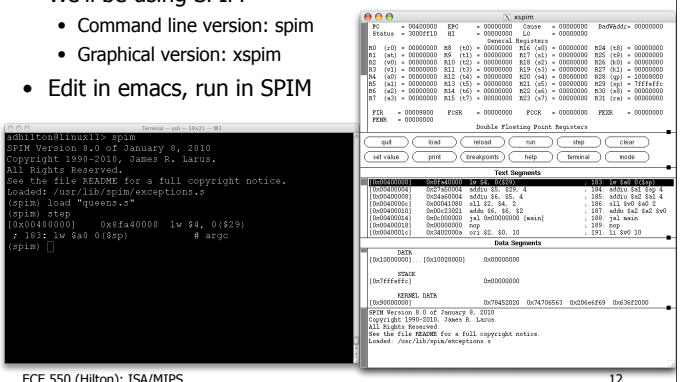
- Instruction Set Architecture (ISA)
  - **Contract between hardware and software**
  - Specifies everything hardware and software need to agree on
    - Instruction encoding and effects
    - Memory endian-ness
    - (lots of other things that won't make sense yet)
- Many different ISAs
  - x86 and x86_64 (Intel and AMD)
  - POWER (IBM)
  - **MIPS**
  - ARM
  - SPARC (Oracle)

Our focus: MIPS

- We will work with MIPS
  - x86 is ugly (x86_64 is less ugly, but still nasty)
  - MIPS is relatively "clean"
    - More on this in a minute

But I don't have a MIPS computer?

- We'll be using SPIM
  - Command line version: spim
  - Graphical version: xspim
- Edit in emacs, run in SPIM



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 mis-conception: 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

Addressing Modes

- Memory location: how to specify address
  - Simple (RISCy)
    - Register + Immediate (e.g., address = r4 + 16)
    - Register + Register (e.g., address= r4 + r7)

Memory Addressing Modes

- Memory location: how to specify address
  - Simple (RISCy)
    - Register + Immediate (e.g., address = r4 + 16)
    - Register + Register (e.g., address= r4 + r7)
  - Complex (CISCy)
    - Auto-increment (e.g., address = r4; r4 = r4 + 4;)
    - Scaled Index (e.g., address = r4 + (r2 <<2) + 0x1234500)
    - Memory indirect (e.g., address = memory[r4])

Load-Store ISA

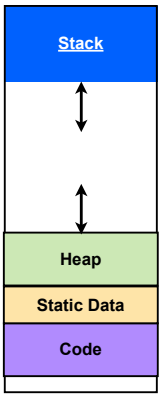
- Load-store ISA:
  - Specific instructions (loads/stores) to access memory
    - Loads read memory (and **only** read memory)
    - Stores write memory (and **only** write memory)
- Contrast with
  - General memory operands (r4 = mem[r5] + r3)
  - Memory/memory operations: mem[r4] = mem[r5] + r3

Stored Program Computer

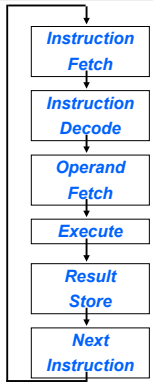
- Instructions: a fixed set of built-in operations
- Instructions and data are stored in memory
  - Allows general purpose computation!
- Fetch-Execute Cycle

```
while (!done)
    fetch instruction
    execute instruction
```

- Effectively what hardware does
- This is what the SPIM Simulator does



How are Instructions 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 #s)
- Result Store:
  Write result to register or memory
- Next Instruction:
  Figure out mem addr of next insn, repeat

More Details on Execution?

- Previous slides high level overview
  - Called von Neumann model
  - John von Neumann: Eniac
- More details: How hardware works
  - Late September
- Now, diving into assembly programming/MIPS

Assembly Programming

- How do you write an assembly program?
- How do you write a program (in general)?

5 Step Plan (ECE 551)

- 5 Steps to write any program:
  1. Work an example yourself
  2. Write down what you did
  3. Generalize steps from 2
  4. Test generalized steps on different example
  5. Translate generalized steps to code

How to Write a Program

- How I teach programming:
  1. Work an example yourself
  2. Write down what you did
  3. Generalize steps from 2
  4. Test generalized steps on different example
  5. Translate generalized steps to code

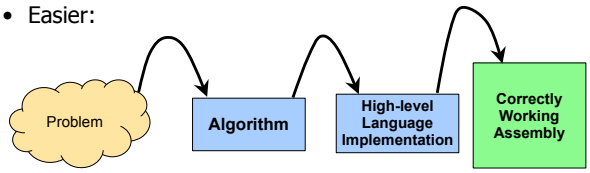
Develop Algorithm In (Familiar) Higher Level Language

Then translate to lower level language

Why do I bring this up?

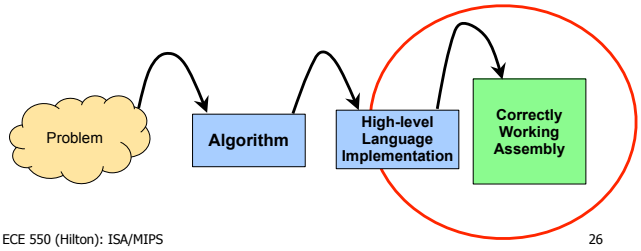
- Very Hard:

- Easier:



Our focus

- We will focus on the assembly step
  - Assume you know how to devise an algorithm for a problem
    - I'll use C.



Simplest Operations We Might Want?

- What is the simplest computation we might do?

Simplest Operations We Might Want?

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

```
x = a + b;
```


Simplest Operations We Might Want?

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

```
x = a + b;
add $r1, $r2, $r3
```

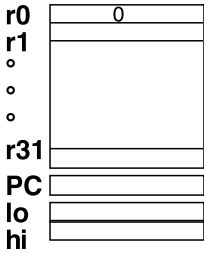
"Add r2 + r3, and store it in r1"

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…

MIPS Integer Registers

- Recall: registers
  - Fast
  - In CPU
  - Directly compute on them
- 31 x 32-bit GPRs (R0 = 0)
- Also FP registers
- A few special purpose regs too
  - PC = Address of next insn



Executing Add

Address	Instruction
1000	add $r8, $r4, $r6
1004	add $r5, $r8, $r7
1008	add $r6, $r6, $r6
100C	…
1010	…

PC tells us where to execute next

Register	Value
R0	0000 0000
R1	1234 5678
R2	C001 D00D
R3	1BAD F00D
R4	9999 9999
R5	ABAB ABAB
R6	0000 0001
R7	0000 0002
R8	0000 0000
R9	0000 0000
R10	0000 0000
R11	0000 0000
R12	0000 0000
…	…
PC	**0000 1000**

Executing Add

Address	Instruction
1000	add $r8, $**r4**, $**r6**
1004	add $r5, $r8, $r7
1008	add $r6, $r6, $r6
100C	…
1010	…

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

9999 9999
0000 0001
9999 999A

Register	Value
R0	0000 0000
R1	1234 5678
R2	C001 D00D
R3	1BAD F00D
R4	**9999 9999**
R5	ABAB ABAB
R6	**0000 0001**
R7	0000 0002
R8	0000 0000
R9	0000 0000
R10	0000 0000
R11	0000 0000
R12	0000 0000
…	…
PC	0000 1000

Executing Add

Address	Instruction
1000	add $**r8**, $r4, $r6
1004	add $r5, $r8, $r7
1008	add $r6, $r6, $r6
100C	…
1010	…

Add writes its result to its destination register

Register	Value
R0	0000 0000
R1	1234 5678
R2	C001 D00D
R3	1BAD F00D
R4	9999 9999
R5	ABAB ABAB
R6	0000 0001
R7	0000 0002
R8	**9999 999A**
R9	0000 0000
R10	0000 0000
R11	0000 0000
R12	0000 0000
…	…
PC	0000 1000

Executing Add

Address	Instruction
1000	add $r8, $r4, $r6
1004	add $r5, $r8, $r7
1008	add $r6, $r6, $r6
100C	…
1010	…

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

Register	Value
R0	0000 0000
R1	1234 5678
R2	C001 D00D
R3	1BAD F00D
R4	9999 9999
R5	ABAB ABAB
R6	0000 0001
R7	0000 0002
R8	9999 999A
R9	0000 0000
R10	0000 0000
R11	0000 0000
R12	0000 0000
…	…
PC	**0000 1004**

Executing Add

Address	Instruction
1000	add $r8, $r4, $r6
1004	add $r5, $r8, $r7
1008	add $r6, $r6, $r6
100C	…
1010	…

You all do the next instruction!

Register	Value
R0	0000 0000
R1	1234 5678
R2	C001 D00D
R3	1BAD F00D
R4	9999 9999
R5	ABAB ABAB
R6	0000 0001
R7	0000 0002
R8	9999 999A
R9	0000 0000
R10	0000 0000
R11	0000 0000
R12	0000 0000
…	…
PC	0000 1004

Executing Add

Address	Instruction
1000	add $r8, $r4, $r6
1004	add $r5, $r8, $r7
1008	add $r6, $r6, $r6
100C	…
1010	…

We set r5 equal to
$r8 (9999 999A) + $r7 (2) = 9999 999C

and PC = PC +4

Register	Value
R0	0000 0000
R1	1234 5678
R2	C001 D00D
R3	1BAD F00D
R4	9999 9999
R5	**9999 999C**
R6	0000 0001
R7	0000 0002
R8	9999 999A
R9	0000 0000
R10	0000 0000
R11	0000 0000
R12	0000 0000
…	…
PC	**0000 1008**

Executing Add

Address	Instruction
1000	add $r8, $r4, $r6
1004	add $r5, $r8, $r7
1008	add $r6, $r6, $r6
100C	…
1010	…

Register	Value
R0	0000 0000
R1	1234 5678
R2	C001 D00D
R3	1BAD F00D
R4	9999 9999
R5	9999 999C
R6	0000 0001
R7	0000 0002
R8	9999 999A
R9	0000 0000
R10	0000 0000
R11	0000 0000
R12	0000 0000
…	…
PC	0000 1008

Its perfectly fine to have $r6 as a src and a dst
This is just like x = x + x; in C, Java, etc:

1 + 1 = 2

Executing Add

Address	Instruction
1000	add $r8, $r4, $r6
1004	add $r5, $r8, $r7
1008	add $r6, $r6, $r6
100C	…
1010	…

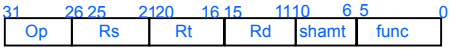
Register	Value
R0	0000 0000
R1	1234 5678
R2	C001 D00D
R3	1BAD F00D
R4	9999 9999
R5	9999 999C
R6	**0000 0002**
R7	0000 0002
R8	9999 999A
R9	0000 0000
R10	0000 0000
R11	0000 0000
R12	0000 0000
…	…
PC	**0000 100C**

Its perfectly fine to have $r6 as a src and a dst
This is just like x = x + x; in C, Java, etc:

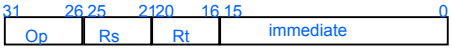
1 + 1 = 2

MIPS Instruction Formats

R-type: Register-Register



I-type: Register-Immediate

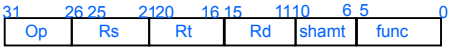


J-type: Jump / Call

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

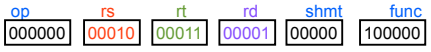
R Type: <OP> rd, rs, rt

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.

Example: add $1, $2, $3

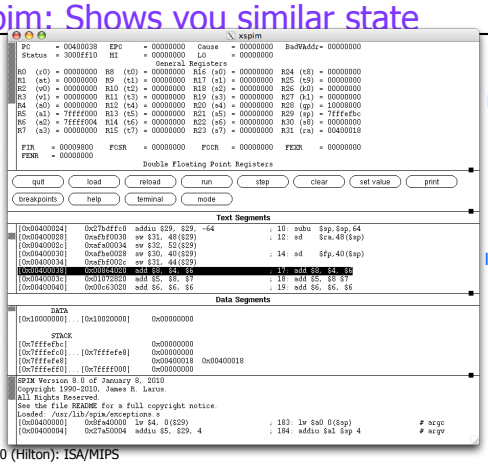


Executing Add

Address	Instruction	Insn Val
1000	add $r8, $r4, $r6	0086 4020
1004	add $r5, $r8, $r7	0107 2820
1008	add $r6, $r6, $r6	00C6 3020
100C	…	…
1010	…	…

Register	Value
R0	0000 0000
R1	1234 5678
R2	C001 D00D
R3	1BAD F00D
R4	9999 9999
R5	9999 999C
R6	0000 0002
R7	0000 0002
R8	9999 999A
R9	0000 0000
R10	0000 0000
R11	0000 0000
R12	0000 0000
…	…
PC	0000 100C

Xspim: Shows you similar state



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
- …

- End of Appendix B: listing of all instructions
  - Good reference, don't need to read every insn
  - Will provide insn reference on tests

Pseudo Instructions

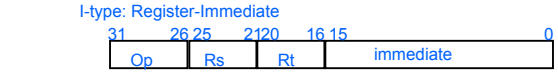
- Some "instructions" are pseudo-instructions
  - Actually assemble into 2 instructions:
- mul $r1, $r2, $r3 is really
  - mul $r2, $r3
  - mflo $r1
- mul takes two srcs, writes special regs lo and hi
- mflo moves from lo into dst reg

What if I want to add a constant?

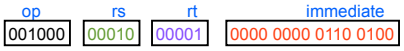
- Suppose I need to do
  - x = x + 1
  - Idea one: Put 1 in a register, use add
    - Problem: How to put 1 in a register?
  - Idea two: Have instruction that adds an **immediate**
    - Note: also solves problem in idea one

I-Type <op> rt, rs, immediate

- Immediate: 16 bit value



Add Immediate Example
addi $1, $2, 100



Using addi to put a const in register

- Can use addi to put a constant into a register:
  - x = 42;
  - Can be done with
  - addi $r7, $r0, 42
  - Because $r0 is always 0.
- Common enough it has its own pseudo-insn:
  - li $r7, 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
  - Why not?
- No muli or divi
  - Though some ISAs have them

Assembly programming something

- Consider the following C fragment:

```
int tempF = 87;          li $r3, 87
int a = tempF - 32;      addi $r4, $r3, -32
a = a * 5;               li $r6, 5
                         mul $r4, $r4, $r6
int tempC = a / 9;       li $r6, 9
                         div $r5, $r4, $r6
```

- Lets write assembly for it
  - First, need registers for our variables:
    - `tempF = $r3`
    - `a = $r4`
    - `tempC = $r5`
  - Now, give it a try (use $r6, $r7,… as temps if you need)…

Accessing Memory

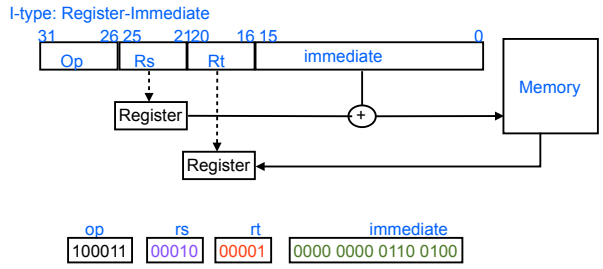
- MIPS is a "load-store" ISA
  - Who can remind us what that means?

Accessing Memory

- MIPS is a "load-store" ISA
  - Who can remind us what that means?
  - Only load and store insns access memory
  - (and that is all those isns do)
- Contrast to x86, which allows
  - add reg = (memory location) + reg
- Or even
  - add (memory location) = (memory location) + reg

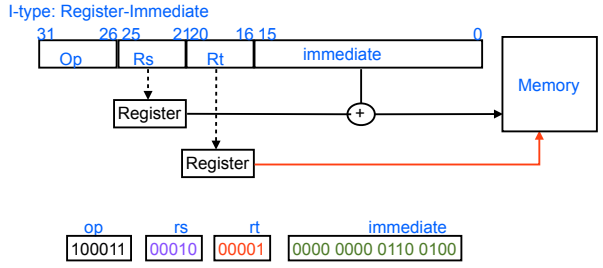
I-Type <op> rt, rs, immediate

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



I-Type <op> rt, rs, immediate

- 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), half words (16 bits)
  - Smaller sizes have signed/unsigned forms
  - See Appendix B for all variants

C and assembly: loads/stores

```
int x                    # in r1
int * p                  # in r2
int ** q                 # in r3
…                        …
x = *p;                  lw $r1, 0($r2)
**q = x;                 lw $r4, 0($r3)
                         sw $r1, 0($r4)
p = *q;                  lw $r2, 0($r3)
p[4] = x;                sw $r1, 16($r2)
```


Executing Memory Ops

Address	Value
1000	lw $r1, 0($r2)
1004	lw $r4, 4($r3)
1008	sw $r1, 8($r4)
100C	lw $r2, 0($r3)
1010	…
8000	F00D F00D
8004	C001 D00D
8008	1234 4321
8010	4242 4242
8014	0000 8000

Register	Value
R0	0000 0000
R1	1234 5678
R2	0000 8004
R3	0000 8010
R4	9999 9999
R5	9999 999C
R6	0000 0002
R7	0000 0002
R8	9999 999A
R9	0000 0000
R10	0000 0000
R11	0000 0000
R12	0000 0000
…	…
PC	0000 1000

Executing Memory Ops

Address	Value
1000	lw $r1, 0($r2)
1004	lw $r4, 4($r3)
1008	sw $r1, 8($r4)
100C	lw $r2, 0($r3)
1010	…
8000	F00D F00D
8004	C001 D00D
8008	1234 4321
8010	4242 4242
8014	0000 8000

Register	Value
R0	0000 0000
R1	C001 D00D
R2	0000 8004
R3	0000 8010
R4	9999 9999
R5	9999 999C
R6	0000 0002
R7	0000 0002
R8	9999 999A
R9	0000 0000
R10	0000 0000
R11	0000 0000
R12	0000 0000
…	…
PC	0000 1004

Executing Memory Ops

Address	Value
1000	lw $r1, 0($r2)
1004	lw $r4, 4($r3)
1008	sw $r1, 8($r4)
100C	lw $r2, 0($r3)
1010	…
8000	F00D F00D
8004	C001 D00D
8008	1234 4321
8010	4242 4242
8014	0000 8000

Register	Value
R0	0000 0000
R1	C001 D00D
R2	0000 8004
R3	0000 8010
R4	0000 8000
R5	9999 999C
R6	0000 0002
R7	0000 0002
R8	9999 999A
R9	0000 0000
R10	0000 0000
R11	0000 0000
R12	0000 0000
…	…
PC	0000 1008

Executing Memory Ops

Address	Value
1000	lw $r1, 0($r2)
1004	lw $r4, 4($r3)
1008	sw $r1, 8($r4)
100C	lw $r2, 0($r3)
1010	…
8000	F00D F00D
8004	C001 D00D
8008	C001 D00D
8010	4242 4242
8014	0000 8000

Register	Value
R0	0000 0000
R1	C001 D00D
R2	0000 8004
R3	0000 8010
R4	0000 8000
R5	9999 999C
R6	0000 0002
R7	0000 0002
R8	9999 999A
R9	0000 0000
R10	0000 0000
R11	0000 0000
R12	0000 0000
…	…
PC	0000 100C

Executing Memory Ops

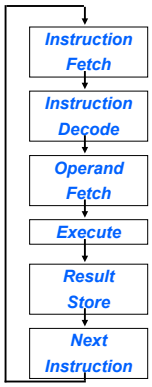
Address	Value
1000	lw $r1, 0($r2)
1004	lw $r4, 4($r3)
1008	sw $r1, 8($r4)
100C	lw $r2, 0($r3)
1010	…
8000	F00D F00D
8004	C001 D00D
8008	C001 D00D
8010	4242 4242
8014	0000 8000

Register	Value
R0	0000 0000
R1	C001 D00D
R2	4242 4242
R3	0000 8010
R4	0000 8000
R5	9999 999C
R6	0000 0002
R7	0000 0002
R8	9999 999A
R9	0000 0000
R10	0000 0000
R11	0000 0000
R12	0000 0000
…	…
PC	0000 1010

Making control decision

- Control constructs—decide what to do next:

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

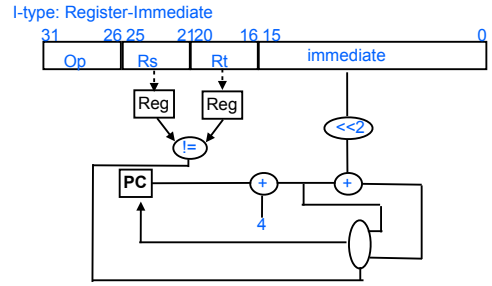


The Program Counter (PC)

- Special register (PC) that points to instructions
- Contains memory address (like a pointer)
- Instruction fetch is
  - inst = mem[pc]
- So far, have fetched sequentially: PC= PC + 4
  - Assumes 4 byte insns
  - True for MIPS
  - X86: variable size (nasty)
- May want to specify non-sequential fetch
  - Change PC in other ways

I-Type <op> rt, rs, immediate

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



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)

MIPS jump, branch, compare

- Inequality to something other than 0: require 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 < $3) ? 1 : 0 $1=0
  - Compare < constant; unsigned
- beqz $1,100 if ($1 == $2) go to PC+4+400
  - Branch if equal to 0
- bnez $1,100 if ($1!= $2) go to PC+4+400

Signed v.s. Unsigned Comparison

- R1= 0…00 0000 0000 0000 0001
- R2= 0…00 0000 0000 0000 0010
- R3= 1…11 1111 1111 1111 1111
- After executing these instructions:
- slt r4,r2,r1
- slt r5,r3,r1
- sltu r6,r2,r1
- sltu r7,r3,r1

- What are values of registers r4 - r7? Why?
- r4 = ; r5 = ; r6 = ; r7 = ;

Signed v.s. Unsigned Comparison

- R1= 0…00 0000 0000 0000 0001
- R2= 0…00 0000 0000 0000 0010
- R3= 1…11 1111 1111 1111 1111
- After executing these instructions:
- slt r4,r2,r1
- slt r5,r3,r1
- sltu r6,r2,r1
- sltu r7,r3,r1

- What are values of registers r4 - r7? Why?
- r4 = 0 ; r5 = 1 ; r6 = 0 ; r7 = 0 ;

C and Assembly with branches

```
int x;                    //assume x in r1
int y;                    //assume y in r2
int z;                    //assume z in r3
…                         …
if (x != y) {             beq $r1,$r2, 2
  z = z + 2;              addi $r3, $r3, 2
}                         b 1
else {
  Z = z -4;               addi $r3, $r3, -4
}
```


Labels

- Counting insns?
  - Error prone
  - Tricky: pseudo-insns
  - Un-maintainable
- Better: let assembler count
  - Use a label
  - Symbolic name for target
  - Assembler computes offset

```
//assume x in r1
//assume y in r2
//assume z in r3
…
   beq $r1,$r2, L_else
   addi $r3, $r3, 2
   b L_end
L_else:
   addi $r3, $r3, -4
L_end:
```


J-Type <op> immediate

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

- Specifies lowest 28 bits of PC
  - Upper 4 bits unchanged
  - Range: 64 Million instruction (256 MB)
- Can jump anywhere with jr $reg (jump register)

Remember our F2C program fragment?

- Consider the following C fragment:

- int tempF = 87; li $r3, 87
- int a = tempF – 32; addi $r4, $r3, -32
- a = a * 5; li $r6, 5
- int tempC = a / 9; mul $r4, $r4, $r6
- li $r6, 9
- If we were really doing this… div $r5, $r4, $r6
  - We would write a function to convert f2c and call it

Remember our f2c fragment?

- Remember this?

```
int tempF = 87;          li $r3, 87
int a = tempF - 32;      addi $r4, $r3, -32
a = a * 5;               li $r6, 5
                         mul $r4, $r4, $r6
int tempC = a / 9;       li $r6, 9
                         div $r5, $r4, $r6
```

If we were really writing this code, we would write a function

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);
```


Need a way to call f2c and return

- Call: Jump… but also remember where to go back
  - May be many calls to f2c() in the program
  - Need some way to know where we were
  - Instruction for this **jal**
  - `jal label`
    - Store PC +4 into register $31
    - Jump to label
- Return: Jump… back to wherever we were
  - Instruction for this **jr**
  - `jr $31`
  - Jump back to address stored by jal in $31

More likely: a function to convert

- 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…

More likely: a function to convert

- 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 pass 87 as argument to f2c

More likely: a function to convert

- 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` to caller

More likely: a function to convert

- 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 use same registers in multiple functions
  - What if f2c called something? 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..
- Alternative: inter-procedural register allocation
  - More work for compiler
  - Only know one real compiler that does this

MIPS Register Naming Conventions

0	zero	constant 0
1	at	reserved for assembler
2	v0	return result
3	v1	(can be used as temp)
4	a0	Argument passing
5	a1	
6	a2	
7	a3	
8	t0	Temporaries
. . .		**Caller saves**
15	t7	
16	s0	**Callee saves**
. . .		
23	s7	
24	t8	Caller saves
25	t9	(continued)
26	k0	reserved for OS
27	k1	kernel
28	gp	Global Pointer
29	sp	Stack pointer
30	fp	frame pointer
31	ra	Return Address

Caller Saves

- Caller saves 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

- Callee saves 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

More likely: a function to convert

```
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

More likely: a function to convert

```
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

More likely: a function to convert

```
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
```


More likely: a function to convert

```
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
```


More likely: a function to convert

```
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

More likely: a function to convert

```
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, $r0, 87
```


More likely: a function to convert

```
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, $r0, 87
 jal f2c
```


More likely: a function to convert

```
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, $r0, 87
 jal f2c
 addi $t0, $v0, 0
```

What it would take to make SPIM

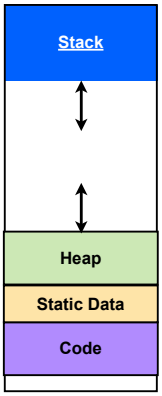
```
      .globl f2c     # f2c can be called from any file
      .ent f2c       # entry point of function
      .text          # goes in "text" region
f2c:                 #    (remember memory picture?)
  addi $t0, $a0, -32
  li $t1, 5
  mul $t0, $t0, $t1
  li $t1, 9
  div $t2, $t0, $t1
  addi $v0, $t2, 0
  jr $ra

    .end f2c         # end of this function
```


Assembly Language (cont.)

- Directives: tell the assembler what to do...
- Format "."<string> [arg1], [arg2] …
- Examples
  - .data [address] # start a data segment.
  - # [optional begining address]
  - .text [address] # start a code segment.
  - .align n # align segment on 2n 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.

The Stack

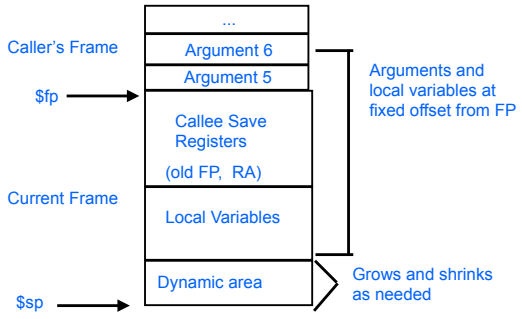
- May need to use the stack for…
  - Local variables whose address is taken
    - (Can't have "address of register")
  - Saving registers
    - Across calls
    - Spilling variables (not enough regs)
  - Passing more than 4 arguments



Stack Layout

- Stack is in memory:
  - Use loads and stores to access
  - But what address to load/store?
- Two registers for stack:
  - Stack pointer ($sp): Points at end (bottom) of stack
  - Frame pointer ($fp): Points at top of current stack frame

Call-Return Linkage: Stack Frames



MIPS/GCC Procedure Calling

- Calling Procedure
- 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 caller-saved registers
  - Save registers $t0-$t9 if they contain live values at the call site.
- Step-3: Execute a jal instruction.
- Step-4: Cleanup stack (if more than 4 args)

MIPS/GCC Procedure Calling

- Called Routine (if any frame is needed)
- Step-1: Establish stack frame.
  - Subtract the frame size from the stack pointer.
    subiu $sp, $sp, <frame-size>
  - Typically, minimum frame size is 32 bytes (8 words).
- Step-2: Save callee saved registers in the frame.
  - Register $fp is always saved.
  - Register $ra is saved if routine makes a call.
  - Registers $s0-$s7 are saved if they are used.
- Step-3: Establish Frame pointer
  - Add the stack <frame size> - 4 to the address in $sp
    addiu $fp, $sp, <frame-size> - 4

MIPS/GCC Procedure Calling

- On return from a call
- Step-1: Put returned values in registers $v0, [$v1].
  (if values are returned)
- Step-2: Restore callee-saved registers.
  - Restore $fp and other saved registers. [$ra, $s0 - $s7]
- Step-3: Pop the stack
  - Add the frame size to $sp.
    addiu $sp, $sp, <frame-size>
- Step-4: Return
  - Jump to the address in $ra.
    jr $ra

Execution example: Calling with frames

Addr	Instruction
1000	subiu $sp, $sp, 16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addiu $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	addiu $sp, $sp, 16
1034	jr $ra

Reg	Value
$r0	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	

Just did jal 1000

Execution example: Calling with frames

Addr	Instruction
1000	subiu $sp, $sp, 16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addiu $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	addiu $sp, $sp, 16
1034	jr $ra

Reg	Value
$r0	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	

$sp, $fp still describe callers frame

Execution example: Calling with frames

Addr	Instruction
1000	subiu $sp, $sp, 16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addiu $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	addiu $sp, $sp, 16
1034	jr $ra

Reg	Value
$r0	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	

Allocate space on stack

Execution example: Calling with frames

Addr	Instruction
1000	subiu $sp, $sp, 16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addiu $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	addiu $sp, $sp, 16
1034	jr $ra

Reg	Value
$r0	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	

Save $fp

Execution example: Calling with frames

Addr	Instruction
1000	subiu $sp, $sp, 16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addiu $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	addiu $sp, $sp, 16
1034	ir $ra

Save $ra

Reg	Value
$r0	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	subiu $sp, $sp, 16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addiu $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	addiu $sp, $sp, 16
1034	ir $ra

Save $s0 (caller saves)

Reg	Value
$r0	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	subiu $sp, $sp, 16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addiu $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	addiu $sp, $sp, 16
1034	ir $ra

Setup $fp

Reg	Value
$r0	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	subiu $sp, $sp, 16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addiu $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	addiu $sp, $sp, 16
1034	ir $ra

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

Reg	Value
$r0	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	subiu $sp, $sp, 16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addiu $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	addiu $sp, $sp, 16
1034	ir $ra

Do some computation..

Reg	Value
$r0	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	subiu $sp, $sp, 16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addiu $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	addiu $sp, $sp, 16
1034	ir $ra

Call another function
(not pictured, takes no args)

jal sets $ra, PC

Reg	Value
$r0	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	

Execution example: Calling with frames

Addr	Instruction
1000	subiu $sp, $sp, 16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addiu $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	addiu $sp, $sp, 16
1034	jr $ra

Reg	Value
$r0	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	subiu $sp, $sp, 16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addiu $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	addiu $sp, $sp, 16
1034	jr $ra

Reg	Value
$r0	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	subiu $sp, $sp, 16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addiu $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	addiu $sp, $sp, 16
1034	jr $ra

Reg	Value
$r0	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	

Do some more computation

Execution example: Calling with frames

Addr	Instruction
1000	subiu $sp, $sp, 16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addiu $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	addiu $sp, $sp, 16
1034	jr $ra

Reg	Value
$r0	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	

Do some more computation
(load addr not pictured)

Execution example: Calling with frames

Addr	Instruction
1000	subiu $sp, $sp, 16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addiu $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	addiu $sp, $sp, 16
1034	jr $ra

Reg	Value
$r0	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	

Restore registers to return

Execution example: Calling with frames

Addr	Instruction
1000	subiu $sp, $sp, 16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addiu $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	addiu $sp, $sp, 16
1034	jr $ra

Reg	Value
$r0	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	

Restore registers to return

Execution example: Calling with frames

Addr	Instruction
1000	subiu $sp, $sp, 16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addiu $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	addiu $sp, $sp, 16
1034	jr $ra

Restore registers to return

Reg	Value
$r0	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	subiu $sp, $sp, 16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addiu $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	addiu $sp, $sp, 16
1034	jr $ra

Restore registers to return

Reg	Value
$r0	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	subiu $sp, $sp, 16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addiu $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	addiu $sp, $sp, 16
1034	jr $ra

Restore registers to return

Reg	Value
$r0	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

Now $sp, $fp describe caller's frame

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	subiu $sp, $sp, 16
1004	sw $fp, 0($sp)
1008	sw $ra, 4($sp)
100C	sw $s0, 8($sp)
1010	addiu $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	addiu $sp, $sp, 16
1034	jr $ra

Return to caller
(code not pictured)

Reg	Value
$r0	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, 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)
    - Live across a call? Use an $s register
    - Otherwise, a $t
  - Write prolog
    - Save ra/fp (if needed)
    - Save any $s registers you use
  - Translate line by line
  - Write epilog

Why do we need FP?

- The frame pointer is not always required
  - May be able to get away without it
- When/why do we need it?
  - Debugging tools (gdb) use it to find frames
  - If you have variable length arrays
    - Stack pointer changes by amount not know at compile time
    - Variables still at constant offset from frame pointer
- 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)
- Load system call code (value) into Register $v0
- Load arguments (if any) into registers $a0, $a1 or $f12 (for floating point).
- do: syscall
- Results returned in registers $v0 or $f0.
- Note: $v0 = $2, $a0=$4, $a1 = $5

SPIM System Call Support

code	service	Arguments	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
8	read	string	$a0=buffer, $a1=length
9	sbrk	$a0=amount	address in $v0
10	exit		

MIPS Assembly General 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], . . .

Summary

- MIPS ISA and Assembly Programming
  - We'll use SPIM (or xspim)
  - Have seen most basic instruction types
  - See Appendix B for full insn reference