

Machine Language: Control Flow and Procedure Calls

◆ Last Time

- Operations + Operands = Computation
- Assembly Language and Machine Language
- Basic Operations and Memory Addressing

◆ This Time

- Control Flow -- jumps and branches
- Memory addressing alternatives
- Assembler directives
- Jumps: long distance control transfers
- JR, JAL Procedure call support, invocation and calling conventions

◆ Announcements

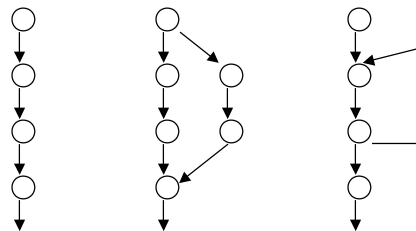
- Read P&H Sections 3.5-3.15
- Also read Appendix A.1-A.6, A.10
- Homework 2, already posted to the Web Pages

CS 141 Chien

1

January 25, 2000

Control Flow



◆ **Idea: Don't always want the same sequence of instructions**

◆ **Conditional execution increases the power and flexibility of computers (more than a calculator)**

◆ **program runs can be much longer than program size (more complex computations)**

CS 141 Chien

2

January 25, 2000

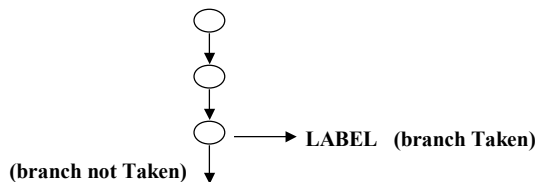
Achieving Conditional Execution

◆ Sequential Instruction Sequence (default)

- Next Instruction = Current Instruction address + 4
- 4 bytes per instruction
- Need another way to choose the next instruction

◆ Conditional Branch Operation

- `beq reg1, reg2, LABEL`
- if `reg1==reg2` Next = LABEL else Next = Current + 4



CS 141 Chien

3

January 25, 2000

Example

```
Loop:  add    $1, $2, $3
       add    $1, $1, $4
       add    $1, $1, $5
       beq    $1, $0, Loop
       ...
```

- ◆ Label -> value for the branch “target”
- ◆ \$0 is always == 0
- ◆ Conditional test
- ◆ This code sequence computes

<u>Conditions</u>	<u>Outcomes</u>
<code>\$2+\$3+\$4+\$5 == 0</code>	Loop forever
otherwise	Loop until <code>\$1 <= 0</code>

CS 141 Chien

4

January 25, 2000

C conditionals and Assembly Language

```
...           ...
if (a != 0) a = a+1;   beq $5, $0, Else
...           addi $5,$5,1
                Else: ...
```

- ◆ Implementing a C conditional (if... else)
- ◆ Suppose `a` is in `$5`
- ◆ True execution path
- ◆ False execution path
- ◆ What if the condition test was more complex?

Conditional Branch Instructions

- ◆ `bne reg1, reg2, Label` **Branch Not Equal**
- ◆ `beq reg1, reg2, Label` **Branch Equal**
- ◆ Complementary tests
- ◆ Only tests for equality
- ◆ What if you want to compare magnitudes?
 - e.g. `A>B`, `A<B`, etc.

Comparing Magnitudes

- ◆ `slt reg1, reg2, reg3` **Set if less than**
 - if `reg3 < reg2`, `reg1 = 1`
 - else `reg1 = 0`
- ◆ `beq` tests value of the comparison
- ◆ **Why doesn't the machine put this all together into condition codes, and compute them for every instruction?**

CS 141 Chien

7

January 25, 2000

Condition codes: an Alternative Approach

```
...           ...
if (A>B) ...   slt  $1,$6,$5
...           bne  $1,$0, ThenClause
                Else: ...
```

- ◆ Every arithmetic instruction sets condition codes such as Zero, NonZero, GreaterThan, Equal, LessThan, Carry, etc.
- ◆ Conditional instructions test these bits
- ◆ Condition codes computed as “side effect” of the arithmetic
- ◆ No additional instructions required to explicitly compute these codes
- ◆ **Problem: getting condition codes right on every instruction complicates hardware design**
- ◆ **Explicit testing doesn't cost much.**

CS 141 Chien

8

January 25, 2000

Another Example: Non-restoring Divide

```
int dividend, divisor, quotient, remainder;
quotient = remainder = 0;

while (dividend > 0){
    quotient = quotient + 1;
    dividend = dividend - divisor;
};
remainder = dividend + divisor;
quotient = quotient - 1;
```

- ◆ Suppose your machine doesn't have a divide instruction
- ◆ Division by repeated subtraction (non-restoring divide)
- ◆ Inefficient, but does compute the correct answer

CS 141 Chien

9

January 25, 2000

Non-restoring Divide (Assembly)

```
                move $3,$0
                move $4,$0
Loop:           slt $5,$1,$0
                bne $5,$0,End
                addi $3,$3,1
                sub $1,$1,$2
                beq $0,$0,Loop
End:           add $4,$1,$2
                subi $3,$3,1
```

Memory Map:

dividend	\$1
divisor	\$2
quotient	\$3
remainder	\$4

- ◆ Basic Parts
- ◆ slt test, beq, fixup

CS 141 Chien

10

January 25, 2000

Memory Addressing

- ◆ `lw reg1, Offset(reg2)`
- ◆ **Displacement Mode**
- ◆ **the ONLY memory addressing mode in MIPS**
- ◆ **reg1 target, Offset = 16 bit displacement, reg2 added to offset**
- ◆ **Many alternative complex modes**
 - Multiple registers
 - Multiple constants
 - Variable size constants
 - Most can be efficiently synthesized from this one primitive

CS 141 Chien

11

January 25, 2000

Immediate Operands

op(6)	rs(5)	rt(5)	constant(16)
-------	-------	-------	--------------

- ◆ `addi $t0, $t0, 4` **Constants in the Instruction**
- ◆ **16 bit constants fit easily**
- ◆ **That's why they use the I format, not R!**
- ◆ **sign extend (copy top bit to upper 16 bits)**
 - 2's complement, 16->32 bit representation, same value
- ◆ **larger constants constructed by shifting, then ORing**

CS 141 Chien

12

January 25, 2000

Addressing in Conditional Branches

beq(6)	rs(5)	rt(5)	Label(16)
--------	-------	-------	-----------

- ◆ `beq reg1, reg2, Label` **Conditional Branch**
- ◆ **How is the label represented?**
 - Label must capture the target instruction's address (32 bits)
- ◆ **But, we only have 16 bits of space.... what should we do?**

Branch Target Addressing

- ◆ **A: Use a relative branch target instruction address**
- ◆ **Target = Program counter + (16 bit constant << 2)**
= a 32-bit address
 - Limits the “distance” one can branch
- ◆ **Example:**

Loop:	Inst	88
	Inst	92
	Inst	96
	beq \$1, \$2, Loop	100
- ◆ **What offset should go in the beq instruction?**

Conditional Branches

- ◆ Execution
 - ◆ Operations, Complex control flow
 - ◆ Relation to C, Loops, etc.
 - ◆ All the pieces, but what is missing?
-
- ◆ How does the assembler decide where to place things? Where to put the program, data, etc?
“Assembler Directives”

CS 141 Chien

15

January 25, 2000

Assembler Directives

- ◆ Labels -- already covered
 - ◆ Instructions -- already covered
- ```
.globl label external labels
.text program to follow this
.data data region to follow

.word 2643 data representation size
.half 43
.byte 6 Beware assembler may
 pack these together
 Alignment directives
.asciiz "stringValue" convenient specification
 of strings
```

CS 141 Chien

16

January 25, 2000

## Typical Program

---

```
.text
.globl ...labels for external stuff...
...
instructions (the program)
...
.data
.word xxx
.word yyy
.word zzz
...
```

- ◆ Program Part
- ◆ Static Data Part
- ◆ Heap allocated things and stack below the static data region

CS 141 Chien

17

January 25, 2000

## Summary so far

---

- ◆ Operations + Operands = Computation
- ◆ But, Operations + Control Flow = A Program
- ◆ Instructions and Assembler Directives together produce an executable assembly program

CS 141 Chien

18

January 25, 2000

## Example: Non-restoring Divide

```
int dividend, divisor, quotient, remainder;
quotient = remainder = 0;

while (dividend > 0){
 quotient = quotient + 1;
 dividend = dividend - divisor;
};
remainder = dividend + divisor;
quotient = quotient - 1;
```

- ◆ Suppose your machine doesn't have a divide instruction
- ◆ Division by repeated subtraction (non-restoring divide)
- ◆ Inefficient, but does compute the correct answer

CS 141 Chien

19

January 25, 2000

## Non-restoring Divide (Assembly)

```
 move $3,$0
 move $4,$0
Loop: slt $5,$1,$0
 bne $5,$0,End
 addi $3,$3,1
 sub $1,$1,$2
 beq $0,$0,Loop
End: add $4,$1,$2
 subi $3,$3,1
```

### Memory Map:

|           |     |
|-----------|-----|
| dividend  | \$1 |
| divisor   | \$2 |
| quotient  | \$3 |
| remainder | \$4 |

- ◆ Basic Parts
- ◆ slt test, beq, fixup

CS 141 Chien

20

January 25, 2000

## Long Distance control transfers

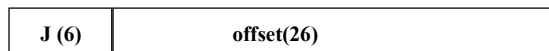
---

```
Loop: ...
 ...
 BEQ $0, $5, Loop
```

- ◆ Conditional branch
- ◆ Opcode(6), 2 register specifiers(5+5), 16 bit offset, + or - 32K instructions (words)
- ◆ What if your loop is too large?
- ◆ What about long distance transfers?
  - additional code layout constraints, “chained branches”
  - multimegabyte images now common

## Jump instructions

---



- ◆ **Solution:** Get a larger offset!
- ◆ How? Special instruction and format.  
    j label
- ◆ 26 bit offset -- 64MB, absolute address, 16MW's
  - => not a relative address, and it's a byte address
- ◆ However, we have a 32-bit address space.
- ◆ What can we do for even larger distances or higher addresses? (rare)

## Jump Register

- ◆ **Answer:** Get an even larger offset!
- ◆ How large is large enough? 32 bits  
jr \$n
- ◆ Jump to address in register
- ◆ 32 bit address, computable target!
- ◆ j and jr are unconditional “Jumps” that support long distance transfers

CS 141 Chien

23

January 25, 2000

## Non-restoring Divide (Assembly)

```
 move $3,$0
 move $4,$0
Loop: slt $5,$1,$0
 bne $5,$0,End
 addi $3,$3,1
 sub $1,$1,$2
 beq $0,$0,Loop => j Loop
End: add $4,$1,$2
 subi $3,$3,1
```

### Memory Map:

|           |     |
|-----------|-----|
| dividend  | \$1 |
| divisor   | \$2 |
| quotient  | \$3 |
| remainder | \$4 |

- ◆ Allows very large loops (even 1M instructions!)
- ◆ What if bne needs a long distance jump?
  - Invert the test, and “skip over” if you want to continue, long jump to the end for exit

CS 141 Chien

24

January 25, 2000

## So far...

---

- ◆ Basic operations (arithmetic and logic)
- ◆ Load/Store (memory)
- ◆ Control transfers
- .... something is missing....
  
- ◆ Can compile/translate any basic C expression or block, but cannot easily implement...

“Procedure Call”

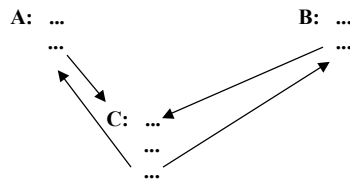
CS 141 Chien

25

January 25, 2000

## Need for Procedure Call Support

---



- ◆ Calls in A and B point to C's entry instruction
- ◆ Where should C's return point to?
- ◆ => must point to a different place for each call
- ◆ => procedure calls require that the target of the return control transfer to be determined dynamically. (no fixed offsets)

CS 141 Chien

26

January 25, 2000

## Building a Return Link

---

- ◆ `jr $n`
- ◆ specifies the target dynamically
- ◆ How to get the value?
  - special macros in assembler
  - load a 32-bit constant which points to the right instruction
  - => convoluted tricks that are confusing and potentially very expensive
- ◆ A single special instruction is the basic support for procedure call/return.

CS 141 Chien

27

January 25, 2000

## Jump and Link

---

- ◆ `jal target`
- ◆ Jumps to target, stores PC+4 (next instruction) in register
- ◆ Where to store the value of return link? (return address)
  - Use register \$31
  - Hardware: stores linked value in \$31
- ◆ This is implicitly specified.
- ◆ Generally cannot use \$31 for other purposes.  
We're down to 30 registers, (`$0 == 0`)

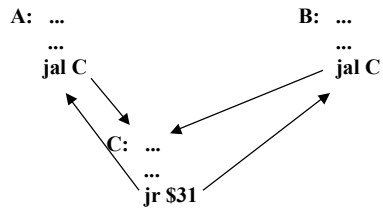
CS 141 Chien

28

January 25, 2000

## Implementing Procedures

---



- ◆ `jal` jumps and puts return link in `$31`
- ◆ `jr $31` jumps to the correct place
- ◆ `jal`, `jr` are sufficient to implement procedure call/return conventions of all flavors.

## More Procedure Calls...

---

- ◆ Is that all there is to procedure calls?
  - Arguments (actuals, formals)
  - Automatics (local variables)
  - Recursion
  - Return values
  - More?

## Example: Procedure Call

---

```
int fibonacci(int i){
 int j,k,l,m;
 float c,d,e,f;
 ...
}

... = fibonacci(6);
```

- ◆ Formals, local variables, actuals
- ◆ Map actuals into formals
- ◆ Allocate space for arguments
- ◆ Stack-based argument passing!
  - But we don't have any stack instructions????

CS 141 Chien

31

January 25, 2000

## Recursive Calls

---

A

|          |
|----------|
| return L |
| return L |
| return L |

- ◆ Are static register allocations likely to work for recursive calls?
- ◆ Is storing the return address in \$31 likely to work for recursive calls?
- ◆ No! Need a stack (Last-in First-out structure)
- ◆ \$29 = stack pointer in MIPS, \$SP works as well!
- ◆ Why? It's just a convention, no special HW support.

CS 141 Chien

32

January 25, 2000

## Pseudo Instructions for Stacks

---

```
Push $X sw $X,0($SP)
 addi $SP,$SP,-4
```

```
Pop $X addi $SP,$SP,4
 lw $X,0($SP)
```

- ◆ Stack grows to lower address (convention)
- ◆ Generally useful primitives, could be used.
- ◆ However stack pointer arithmetic is usually optimized (producing slightly more complex code).

## Stack Pointer Arithmetic Optimization

---

```
Push $10 sw $10,0($SP)
 addi $SP,$SP,-4
Push $11 sw $11,0($SP)
 addi $SP,$SP,-4
Push $12 sw $12,0($SP)
 addi $SP,$SP,-4
```

- ◆ Each stack operation -> 2 instructions
- ◆ Intermediate value of \$SP no visible to others
- ◆ Consolidate arithmetic, make sure offsets are correct.

## Stack Pointer Arithmetic Optimization

---

|           |                                      |                                        |
|-----------|--------------------------------------|----------------------------------------|
| Push \$10 | sw \$10,0(\$SP)<br>addi \$SP,\$SP,-4 | sw \$10,0(\$SP)                        |
| Push \$11 | sw \$11,0(\$SP)<br>addi \$SP,\$SP,-4 | sw \$11,-4(\$SP)                       |
| Push \$12 | sw \$12,0(\$SP)<br>addi \$SP,\$SP,-4 | sw \$12,-8(\$SP)<br>addi \$SP,\$SP,-12 |

- ◆ Consolidate arithmetic, make sure offsets are correct. Works for both Push and Pop.
- ◆ 6 -> 4 instructions, Much faster!
- ◆ Oscillations, no arithmetic at all!

## Procedure Call (for real)

---

- ◆ Push return addresses on the stack
- ◆ allocate local storage for automatic variables on the stack
- ◆ arguments?
- ◆ Basic: Pass arguments in the registers, but if not enough registers, pass arguments on the stack.
- ◆ Why? Registers are generally much faster than the stack (which is in the main memory).

## C Calling Conventions (for MIPS)

---

- ◆ 4 arguments in \$4-\$7
- ◆ Refer to these as \$a0 - \$a3, Arguments 0-3
- ◆ + a bunch of other register usage conventions
  
- ◆ => First, a simple example.

CS 141 Chien

37

January 25, 2000

## Basic C Call

---

```
A: ...
 ...
 move $a0,$9 # a better program
 move $a1,$10 # might compute these values
 move $a2,$11 # into place
 move $a3,$12
 jal my_procedure
```

my\_procedure:

```
sw $31,0($SP)
addi $SP,$SP,-32
...
jal my_procedure
...
addi $SP,$SP,32
lw $31,0($SP)
jr $31
```

The diagram shows a call from 'A' to 'my\_procedure' and a return from 'my\_procedure' back to 'A'. An arrow points from the 'jal my\_procedure' instruction in the first code block to the 'my\_procedure:' label in the second code block. Another arrow points from the 'jr \$31' instruction in the second code block back to the 'A:' label in the first code block.

- ◆ Basic Structure
- ◆ Stack setup, return links
- ◆ Recursive call (conditional)
- ◆ => regs must be saved over calls, HOW?

CS 141 Chien

38

January 25, 2000

## Full MIPS Calling Conventions

---

|           |                      |                                     |
|-----------|----------------------|-------------------------------------|
| \$0       | Constant 0           |                                     |
| \$AT      | \$1                  | Reserved for Asm and OS             |
| \$v0-\$v1 | \$2-\$3              | Return Values                       |
| \$a0-\$a3 | \$4-\$7              | Arguments                           |
| \$t0-\$t9 | \$8-\$16, \$24, \$25 | Caller saved Registers              |
| \$s0-\$s7 | \$16-\$23            | Callee saved Registers              |
| \$k0-\$k1 | \$26-\$27            | Reserved for Asm and OS             |
| \$gp      | \$28                 | Global Pointer                      |
| \$sp      | \$29                 | Stack Pointer                       |
| \$fp      | \$30                 | Frame Pointer (base of stack frame) |
| \$31      |                      | Return address from jal             |

CS 141 Chien

39

January 25, 2000

## Caller/Callee Saved

---

|                               |                               |
|-------------------------------|-------------------------------|
| <u>Caller Saved</u> \$t0-\$t9 | <u>Callee Saved</u> \$s0-\$s7 |
| ... # values okay here        | ... #values okay              |
| call ...                      | call ...                      |
| ... # trashed here            | ... #values still okay        |

- ◆ Determines who's responsible for preserving the state, and what can be used without concern.
- ◆ Caller saved: Values don't persist over a call, save these before a call if you want to use.
  - Callee can use these without saving them.
- ◆ Callee saved: Values persist.
  - Callee must save these and restore after done.
- ◆ What does all this mean?
  - Work shared between caller and callee.
  - Do a little as possible.
  - Where's the state of one call? Distributed over the stack.

CS 141 Chien

40

January 25, 2000

## Summary

---

- ◆ Basic operations (arithmetic and logic)
- ◆ Load/Store (memory)
- ◆ Control transfers
- ◆ `j`, `jr`, `jal` long distance control transfers
- ◆ `jal` and `jr` are the key to procedure calls
  - Stacks
  - Register usage/calling conventions