CSE 30321 – Lecture 07-08 – In Class Example Handout

Part A: J-Type Example:

If you look in your book at the syntax for j (an unconditional jump instruction), you see something like:

j addr
e.g. would seemingly say, PC ← addr

But, this actually isn't very realistic:

Format of jump is: I 6-bit opcode I 26-bit target address I

Only 26 bits would leave many addresses unreachable:
$2^{32}/2^2 - 2^{26}/2^2$ = 1,056,964,608 specifically

In reality, jump allows you to go "farther" then you can with 16 bits of offset
(2^{26} / 2^2 → you can move ~16M instructions from where you're currently at)

But, for sake of completeness, what does happen is this:
New address: PC(31:28) **I** 26 bits of target address **I** 00 (to align to word)

Note: in HW, etc. j "else" perfectly OK unless noted.

Also, to compare to i-Type branch...

- beq $6, $7, else
  - beq = 6 bits
  - $6 = 5 bits
  - $7 = 5 bits
  - else = 16 bits
- PC ← PC + 4 + (offset associated with else x 4)

Note, again, beq $6, $7, else is OK.

Part B: I-Type and Endianness:

Part 1: Endianness

Byte Ordering

- The *layout* of multi-byte operands in memory must also be carefully considered.
- There are 2 ways that data might be "laid out"/arranged in a given memory location.
  - Little Endian: Least significant byte is at the lowest address in memory. (This is the way that x86 does things...)
  - Big Endian: Most significant byte is at the lowest address in memory. (This is the way that "everything else" does things...)
  - (We'll assume Big Endian unless otherwise noted...)
- For example, here is a hex representation of a 32 bit dataword: AA|BB|CC|DD. The most significant bit is A and the least significant bit is D.
  - In Little Endian, the most significant part of the data word would be at the most significant address and the least signficant part of the data word would be at the least signficant part of the address.
    * i..e $DD_{00}|CC_{01}|BB_{10}|AA_{11}$
    * This is probably the most "intuition friendly"
  - In Big Endian the situation is reversed – i.e. $AA_{00}|BB_{01}|CC_{10}|DD_{11}$. This is more "reader friendly."

(Class question?)

Here's another example. Assume the 4-byte word 0x11223344 is stored at word address 100. How is the data distributed in Big and Little Endian forms?

Big Endian:

- (Byte address in first row, word address = 100, MSB at word address)

Table 1: Big Endian

100	101	102	103
11	22	33	44
+0	+1	+ 2	+3

Little Endian

- (LSB at word address)

Table 2: Little Endian

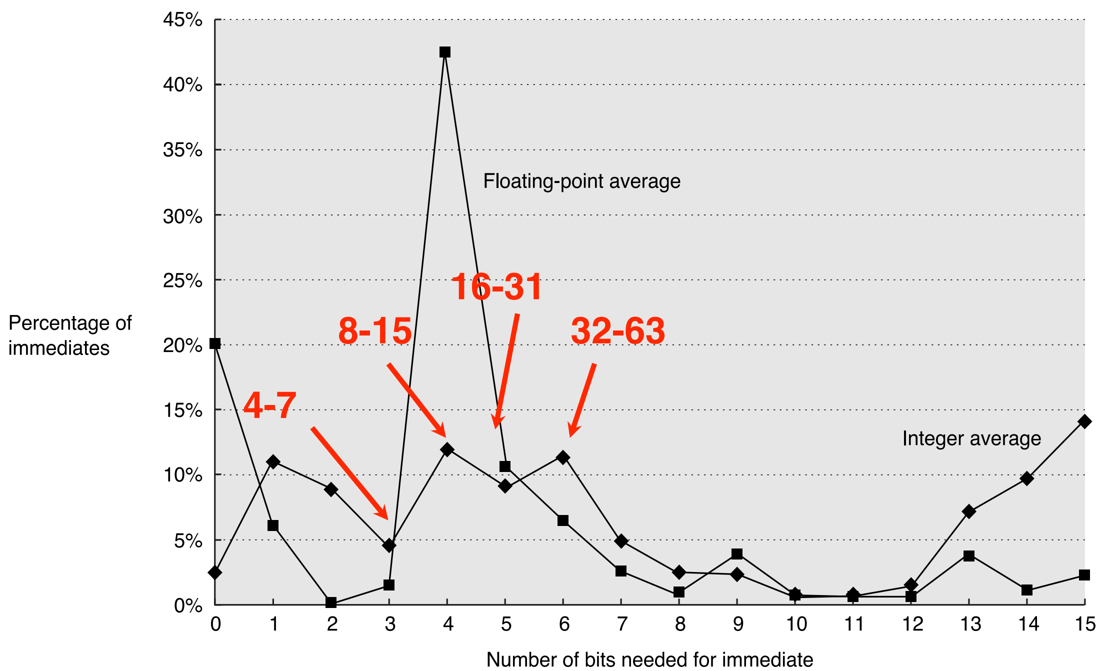
103	102	101	100
11	22	33	44
+3	+2	+ 1	+0

Important if sending data from 1 machine to the another. (e.g. x86 → AMD, Power PC)

Part 2: I-Type Constants

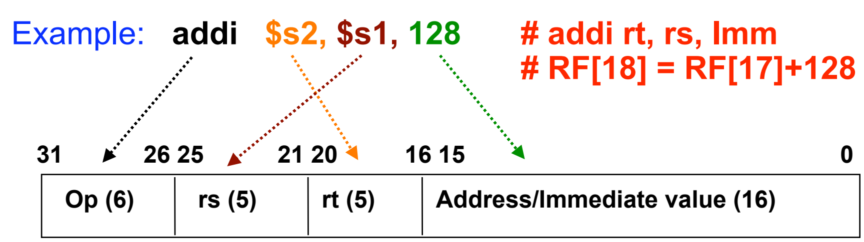
What does the graph below – the size of common immediate (hard coded constant) values – tell you about the numbers of bits that have been made available for a hard coded constant in a MIPS immediate instruction?

Frequency of constant requiring N bits:



I-Type Encoding:

- **I-type: One operand is an immediate value and others are in registers**



B: 001000 10001 10010 0000000010000000

D: 8 17 18 128

Answer:

Part C: A Simple, MIPS-based Procedure:

Swap Procedure Example:
Let's write the MIPS code for the following statement (and function call):

```
if (A[i] > A [ i+1])          // $s0 = A
    swap (&A[i], &A[i+1])     // $t0 = 4*i
```

We will assume that:
- The address of A is contained in $s0 ($16)
- The index (4 x i) will be contained in $t0 ($8)

Answer:

The Caller:

```
        …
        // Calculate address of A(i)
        add     $s1, $s0, $t0       // $s1 ← address of array element i in $s1

        // Load data
        lw      $t2, 0($s1)         // load A(i) into temporary register $t2
        lw      $t3, 4($s1)         // load A(i+1) into temporary register $t3

        // Check condition
        ble     $t2, $t3, else      // is A(i) <= A(i+1)?  If so, don't swap

        // if >, fall through to here…
        addi    $a0, $t0, 0         // load address of x into argument register (i.e. A(i))
        addi    $a1, $t0, 4         // load address of y into argument register (i.e. A(i+1))

        // Call Swap function
        jal     swap                // PC ← address of swap; $ra l $31 = PC + 4
else:
```

// Note that swap is "generic" – i.e. b/c of the way data is passed in, we do not assume the values
// to be swapped are in contiguous memory locations – so two distinct physical addresses are
// passed in

Swap:

```
        lw $t0, 0($a0)      // $t0 = mem(x) – use temporary register
        lw $t1 0($a1)       // $t1 = mem(y) – use temporary register
        sw 0($a0), $t1      // do swap
        sw 0($a1), $t0      // do swap
        jr $ra              // $ra should have PC = PC + 4
                            // PC = PC + 4 should be the next address after jump to swap
```

Part D: Procedures with Callee Saving (old exam question):

Assume that you have written the following C code:

```
//-------------------------------------------------------------------
int   variable1   = 10;                   // global variable
int   variable2   = 20;                   // global variable
//-------------------------------------------------------------------
int main(void) {
      int i       = 1;                    // assigned to register s0
      int j       = 2;                    // assigned to register s1
      int k       = 3;                    // assigned to register a3
      int m;
      int n;

      m = addFourNumbers(i, j);

      n = i + j;                          // 1 + 2    = 3

      printf("m is %d\n", m);             // printf modifies no registers
      printf("n is %d\n", n);             // printf modifies no registers
      printf("k is %d\n", k);             // printf modifies no registers
}
//---------------------------------------------------------------------
int addFourNumbers(int x, int y) {
      int i;                              // assigned to register s0
      int j;                              // assigned to register s1
      int k;                              // assigned to register s2

      i     = x + y;                      // 1 + 2    = 3
      j     = variable1 + variable2;      // 10 + 20  = 30
      k     = i + j;                      // 3 + 30   = 33

      return k;
}
//---------------------------------------------------------------------

The output of the printf statements in main is: m is 33
                                                n is 3
                                                k is 3
```

Assume this program was compiled into MIPS assembly language with the register conventions described on Slide 12 of Lecture 07/08. Also, note that in the comments of the program, I have indicated that certain variables will be assigned to certain registers when this program is compiled and assembled. Using a callee calling convention, answer the questions below:

Q-i: Ideally, how many <u>arguments</u> to the function addFourNumbers must be saved on the stack?

0. By default, arguments should be copied into registers.

Q-ii: What (if anything) should the assembly language for main() do right before calling addFourNumbers?

Copy values of s registers into argument registers; save value of k (in $a3) onto the stack

Q-iii: What is the first thing that the assembly language for addFourNumbers should do upon entry into the function call?

Callee save the s registers

Q-iv: What is the value of register number 2 (i.e. 0010_2) after main completes (assuming there were no other function calls, no interrupts, no context switches, etc.)

33. Register 2 = v0. It should *not* have changed.
(different answer if you assume printf returns value)

Q-v: Does the return address register ($ra) need to be saved on the stack for this program? Justify your answer. (Assume main() does not return).

No – if no other procedures are called.

Part E: Procedures with Callee Saving (old exam question):

Assume you have the following C code:

```
int main(void) {
        int x = 10;                 # x maps to $s1
        int y = 20;                 # y maps to $s2
        int z = 30;                 # z maps to $s3

        int a;                      # a maps to $t0
        int b;                      # b maps to $t1
        int c;                      # c maps to $t2

        c = x + y;

        a = multiply(x, z);

        b = c + x;
}

int multiply(int a, int b) {
        int q;                      # q maps to $t0
        int z;          ■           # z maps to $t1

        q = add();
        z = a*b;            ▲

        return z;
}

int add() {
        int m = 5;                  # m maps to $t4
        int n = 4;                  # n maps to $t5
        int y;
        y = a + b;
        return y;
}
```

Assuming the MIPS calling convention, answer questions A-E. Note – no assembly code/machine instructions are required in your answers; simple explanations are sufficient.

Q-i: What, if anything must main() do before calling multiply?

- **Save $t2 to stack, needed upon return.**
- **Also, copy $s1 to $a0 and copy $s3 to $a1**

Q-ii: Does multiply need to save anything to the stack? If so, what?

- **$31**
- **The s registers associated with main()**
- **The argument registers passed into multiply before calling add()**

Q-iii: Assume that multiply returns its value to main() per the MIPS register convention. What machine instructions might we see at ▲ to completely facilitate the function return?

- **We have to copy the value in $t1 to $2.**
- **We would call a jal instruction**
- **We would adjust the stack pointer, restored saved registers.**

Q-iv: What line of code should the return address register point to at ▲?

- **b = c + x**

Other answers were considered correct based on stated assumptions.

Q-v: What line of code should the return address register point to at ■?

- **b = c + x**

Other answers were considered correct based on stated assumptions.

Part F: More Complex Example:

Let's write the MIPS code for the following:

```
for(i=1; i<5; i++) {
  A(i) = B*d(i);
  if(d(i) >= e) {
    e = function(A,i);
  }
}
```

```
int function(int, int) {
  A(i) = A(i-1);
  e = A(i);
  return e;
}
```

```
Assume:
Addr. of A = $18
Addr. of d = $19
B = $20
e = $21
```

```
(We pass in starting "address of A" and "i")
```

Question/Comment	My Solution	Comment
1st, want to initialize loop variables. **What registers should we use, how should we do it?**	addi $16, $0, 1 addi $17, $0, 5	# Initialize i to 1 # Initialize $17 to 5 (in both cases, *saved* registers are used – we want this data available post function call)

2nd, calculate address of d(i) and load. **What kind of registers should we use?**	Loop: sll $8, $16, 2
	add $8, $19, $8
	lw $9, 0($8)
		# store i*4 in $8 (temp register OK)
		# add start of d to i*4 to get address of d(i)
		# load d(i) → needs to be in register to do math

Calculate B*d(i)	mult $10, $9, $20	# store result in temp to write back to memory

Calculate address of A(i)	sll $11, $18, 2
	add $11, $11, $18
	
	CANNOT do:
	add $11, $8, $18
		# Same as above
		
		# We overwrote
		# But, would have been better to save i*4
		Why? Lower CPI

Store result into A(i)	sw 0($11), $10	# Store result into a(i)
Now, need to check whether or not d(i) >= e. **How? Assume no ble.**	slt $1, $9, $22 bne $0, $1, start again	# Check if $9 < $22 (i.e. d(i) < e) # Still OK to use $9 → not overwritten # (temp does not mean goes away immediately) # if d(i) < e, $1 = 1 # if d(i) >= e, $1 = 0 (and we want to call function) # (if $1 != 0, do not want to call function)

Given the above setup, what comes next? (Falls through to the next function call). **Assume argument registers, what setup code is needed?**	add $4, $18, $0
	add $5, $16, $0
	x: jal function
		# load address of (A) into an argument register
		# load i into an argument register
		# call function; $31 ← x + 4 (if x = PC of jal)

Finish rest of code: **What to do?** Copy return value to $21. Update counter, check counter. **Where is "start again" at?**	add $21, $0, $2 sa: addi $16, $16, 1 bne $16, $17, loop	# returned value reassigned to $21 # update i by 1 (array index) # if i < 5, loop A better way: Could make array index multiple of 4
Function Code		

Assume you will reference A(i-1) with lw … 0($x). **What 4 instruction sequence is required?**	func: subi $5, $5, 1
	sll $8, $5, $2
	add $9, $4, $8
	lw $10, 0($9)
		# subtract 1 from i
		# multiply i by 4 → note
		# add start of address to (i-1)
		# load A(i-1)

Finish up function.	sw 4($9), $10 add $2, $10, $0	# store A(i-1) in A(i) # put A(i-1) into return register ($2)
Return	jr $31	# PC = contents of $31

Part G: Nested Function Calls

```
int main(void) {
  i = 5;   # i = $16
  j = 6;   # j = $17
  k = foo1();
  j = j + 1;
}
```

```
foo1() {
  a = 17;   # a = $16
  b = 24;   # b = $17
  …
  foo2();
}
```

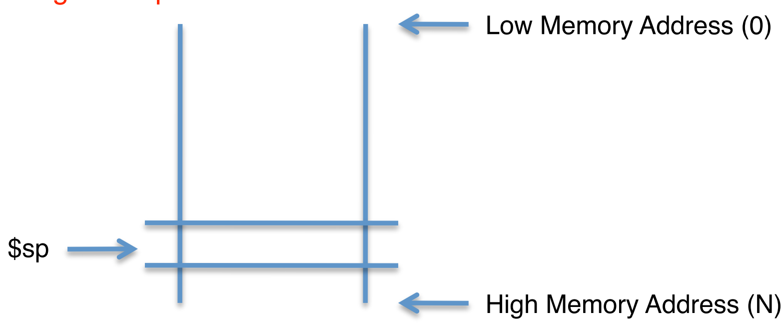
```
foo2() {
  x = 25; # x = $16
  y = 12; # y = $17
}
```

Let's consider how we might use the stack to support these nested calls.

Question:
How do we make sure that data for i, j ($16, $17) is preserved here?

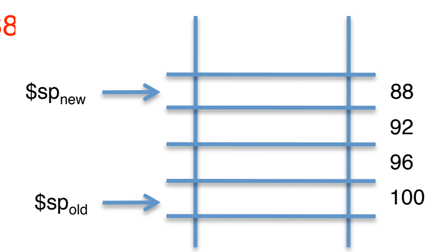
Answer:
Use a stack.

By convention, the stack grows up:



Let's look at main():

- Assume we want to save $17 and $16
  - (we'll use the stack pointer)
- Also, anything else we want to save?
  - $31 – if nested calls.
- How?
  - subi $sp, $sp, 12 # make space for 3 data words
  - Example: assume $sp = 100, therefore $sp = 100 – 12 = 88
- Then, store results:
  - sw 8($sp), $16 # address: 8 + $sp = 8 + 88 = 96
  - sw 4($sp), $17 # address: 4 + $sp = 4 + 88 = 92
  - sw 0($sp), $31 # address: 0 + $sp = 0 + 88 = 88



Now, in Foo1() … assume A and B are needed past Foo2() … how do we save them?

- We can do the same as before
  - Update $sp by 12 and save

Similarly, can do the same for Foo2()

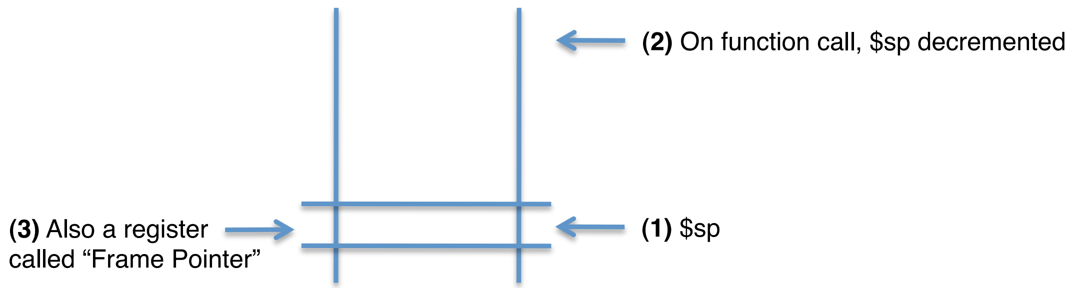
Now, assume that we are *returning* from Foo1() to main(). What do we do?

- The stack pointer should equal the value before the Foo1() call (i.e. 88)

```
lw $31, 0($sp)     # $31 ← memory(0 + 88)     (LIFO)
lw $17, 4($sp)     # $17 ← memory(4 + 88)
lw $16, 8($sp)     # $16 ← memory(8 + 88)
```

Finally, update $sp: addi $sp, $sp, 12 ($sp now = 100 again)

Let's talk about the Frame Pointer too:

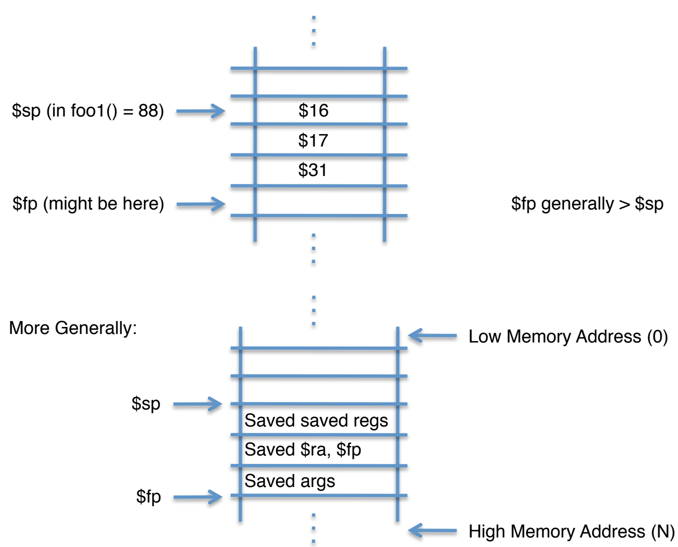


$fp (frame pointer) points to the "beginning of the stack" (ish) – or the first word in frame of a procedure

Why use a $fp?

- Stack used to store variables local to procedure that may not fit into registers
- $sp can change during procedure (e.g. as just seen)
    - Results in different offsets that may make procedure harder to understand
- $fp is stable base register for local memory references

For example:



-Therefore procedure might reference extra function argument as 0($fp)
-What if 2 saved arguments? What next?
-With this convention: lw $t0, 4($fp)

Because $sp can change dynamically, often easier/intuitive to reference extra arguments via stable $fp – although can use $sp with a little extra math

Part H: Recursive Function Calls

```
int fact(int n) {
     if (n<1)
          return(1);
     else
          return(n*fact(n-1));
}
```

Let's consider how we might use the stack to support these nested calls. We'll also make use of the frame pointer ($fp).

```
1:   Fact:  subi $sp, $sp, 12          # make room for 3 pieces of data on the stack –
                                       # $fp, $sp, 1 local argument
                                       # Therefore, if $sp = 100, its now 88
            sw 8($sp), $ra             # M(88 + 8) ← $ra     (store return address)
            sw 4($sp), $fp             # M(88 + 4) ← $fp     (store frame pointer)
            subi $fp, $fp, 12          # update the frame pointer
                                       # - could assume its 1 above old $sp
                                       # - book uses convention here (i.e. $sp = $fp)
                                       # - therefore return to data at 0($fp)
                                       # - in the other case, it would be 4($fp)

2:          bgtz $a0, L2               # if N > 0 (i.e. not < 1) we're not done
                                       # we assume N is in $a0

4:          addi $v0, $0, 1            # we eventually finish and want to return 1
                                       # put 1 in return register
            j L1                       # jump to return code

3:   L2:    sw $a0, 0($fp)       ***   # save argument N to stack (we'll need it when we return)
            subi $a0, $a0, 1           # decrement N (N = N – 1), put result in $a0
            jal Fact                   # call Factorial() again

6:   @      lw $t0, 0($f0)             # load N (saved at *** to stack)
            mult $v0, $v0, $t0         # store result in $v0

5:   L1     lw $ra, 8($sp)             # restore return address
            lw $fp, 4($sp)             # restore frame pointer
            addi $sp, $sp, 12          # pop stack
            jr $ra                     # re
```

More specific, quantitative example:

84		
88		← $\$sp_{new}$
92	$fp	
96	$ra	
100	N	← $\$sp_{old}$
104		
108		
112	N+1	← $\$fp_{old}$
116		

Assume $sp initially is equal to 100. Therefore:

```
subi    $sp, $sp, 12        # $sp = 88
sw      $ra, 8($sp)         # M(88 +8) ← $ra
sw      $fp, 4($sp)         # M(88 +4) ← $fp
subi    $fp, $fp, 12        # $fp = 112 – 12 = 100
bgtz
sw      $a0, 0($fp)         # M(100) ← N (i.e. $ao)
```

On return:

- Assume N < 1

```
5:  lw      $ra, 8($sp)     #   $ra ← M(8 + 88)
    lw      $fp, 4($sp)     #   $fp ← M(4 + 88)
                                $fp now 112…
    subi    $sp, $sp, 12    #   $sp = 100 again

6:  lw      $t0, 0($fp)     #   $t0 ← M($fp) OR $ t0 ← M(112)
                                (which, per convention, would be N+1 from previous
                                call; use to start multiply accumulation)
```