CSE 30321 – Computer Architecture I – Fall 2009
Lecture 09– In Class Examples
September 22, 2009

Question 1:

```
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

Question 2:

```
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.

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

Question 3:

```
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                     # return (to @)
```