hw7.md

EECS 645 - HW 7

Author: Jace Kline

Problem 2.24

PC = 0x2000 0000, target = 0x4000 0000

No, no. The jump instruction (j) has an address field of 26 bits. After this value is shifted left by 2 bits, this value is multiplied by 4 and extends the value to 28 bits. We concatenate the first 4 bits from the PC address to the left of this address (most significant bits), which results in a 32-bit absolute address. Since the final jump destination address must inherit the furthest left byte from the PC address, the jump destination must be of the form jump_dest = 0x2000 0000 + offset, where offset <= 0x0fff fffc. Hence, jump_dest >= 0x2000 0000 and jump_dest <= 0x2fff fffc. As we can see, for all jump addresses possible, jump_dest < target and therefore it is not possible for a jump to reach 0x4000 0000. Since the branch-on-equal (beq) has only a 16-bit address field which is less than the jump range, it is additionally not possible to reach the target address via a 'beq' instruction.

Problem 2.25

2.25.1

This instruction would be an I-format instruction. This is because it is a variation of a conditional branching instruction similar to 'beq'.

2.25.2

Code (without pseudo instruction 'ble'):

```
slt $t0, $zero, $t2   # $t0 = ($t2 > 0) ? 1 : 0
beq $t0, $zero, loop  # if($t0 == 0) goto loop
addi $t2, $t2, -1     # $t2 = $t2 - 1
```

Code (with pseudo instruction 'ble'):

```
ble $t2, $zero, loop  # if($t2 <= 0) goto loop
addi $t2, $t2, -1     # $t2 = $t2 - 1
```

Problem 2.26

2.26.1

Assume $t1 = 10, $s2 = 0. There are 10 iterations of the loop and each iteration increments the value in $s2 by 2. Hence, we get `$s2 = 10(2) + 0 = 20`. `Answer: $s2 = 20`

2.26.2

Let registers $s1, $s2, $t1, and $t2 correspond to the C integer variables A, B, i, and temp. Equivalent C code:

```
int i;
int B;
```

```
// ... initialize i and B ...
while(i > 0) {
  i = i - 1;
  B = B + 2;
}
```

2.26.3

Suppose $t1 stores value N for some N >= 0. The case when i = 0 causes 2 instructions to execute. The case when i > 0 (one iteration) causes 5 instructions to execute. Given i = N, we will execute N full iterations followed by the i = 0 case. Therefore:

```
Instructions executed = 5N + 2
```

Problem 2.27

Assume values a, b, i, and j are in registers $s0, $s1, $t0, and $t1. Also assume $s2 holds the value for the address of array D.

```
add $t0, $zero, $zero       # initialize i=0 for the outer loop

# Lbl: Loop_Outer
slt $t3, $t0, $s0           # let t3 = (i < a) ? 1 : 0
beq $t3, $zero, Exit_Outer  # if t3 is false, branch to Exit_Outer label address
add $t1, $zero, $zero       # initialize j=0 for the inner loop

# Lbl: Loop_Inner
slt $t3, $t1, $s1           # let t3 = (j < b) ? 1 : 0
beq $t3, $zero, Exit_Inner  # if t3 is false, branch to Exit_Inner label address
sll $t3, $t1, 2             # let t3 = (4 * j)
add $t3, $s2, $t3           # let t3 = (address of array D) + (offset of value in t3)
add $t4, $t0, $t1           # let t4 = i + j
sw $t4, 0($t3)              # store value t4 in memory at address stored in t3
addi $t1, $t1, 1            # j = j + 1
j Loop_Inner                # jump back to the start of the inner loop

# Lbl: Exit_Inner
addi $t0, $t0, 1            # i = i + 1
j Loop_Outer                # jump back to the start of the outer loop

# Lbl: Exit_Outer
...
```

Problem 2.31

Part 1: Implement fib in MIPS assembly

We assume that $v0 is the return value register and the $a0 register shall hold the argument n on each recursive call. We assume that we are not required to save and restore the value of temporary register $t0 when we return.

```
# Lbl: fib_0                # Base Case (n == 0)
bne $a0, $zero, fib1        # if (n != 0) goto fib_1
add $v0, $zero, $zero       # otherwise, store return value 0 into $v0
jr $ra                      # jump back to return address

# Lbl: fib_1                # Base Case (n == 1)
addi $t0, $zero, 1          # let t0 = 1
bne $a0, $t0, fib_rec       # if (n != 1) goto fib_rec
addi $v0, $zero, 1          # otherwise, store return value 1 into $v0
jr $ra                      # jump back to return address
```

```
# Lbl: fib_rec              # Recursive Case (n > 1)
addi $sp, $sp, -12          # Allocate a stack frame to hold the caller's argument $a0 and caller's return address $ra
sw $ra, 8($sp)              # Store $ra on the stack
sw $a0, 4($sp)              # Store $a0 on the stack
sw $s0, 0($sp)              # Store $s0 on the stack (we use this in our computation)
addi $a0, $a0, -1           # Decrement the argument n for our first recursive call (n = n - 1)
jal fib_0                   # store next instruction's address in $ra and jump (recursive) to the start of the procedure ->
calling fib(n-1)
add $t0, $v0, $zero         # let t0 = return value of fib(n-1)
addi $a0, $a0, -1           # Decrement argument again (n = n - 1)
jal fib_0                   # jump and link -> calling fib(n-2)
add $v0, $t0, $v0           # return value = fib(n-1) + fib(n-2)
lw $s0, 0($sp)              # reload the original value of $s0
lw $a0, 4($sp)              # reload argument (n) from stack into $a0
lw $ra, 8($sp)              # reload caller's return address from the stack into $ra
addi $sp, $sp, 12           # reset the stack pointer to last frame
jr $ra                      # return to caller
```

Part 2: Number of MIPS instructions

Suppose we are given some n where n >= 0. We must compute the number of MIPS instructions it will take to compute fib(n).

```
Let y(n) be the function that measures the number of MIPS instructions as a function on n.
We can find our initial conditions as follows:
y(0) = instructions for base case fib(0)
y(0) = 3

y(1) = instructions for base case fib(1) = 1 (from fib_0) + 4 (from fib_1)
y(1) = 5

y(2) = instructions for fib(2) = y(0) + y(1) + (1 (from fib_0) + 2 (from fib_1) + 15 (from fib_rec))
y(2) = 26

Inductive Case (n > 1):
y(n) = 18 + y(n-1) + y(n-2)

We must eliminate the constant term 16 from this recurrence relation in order to find a characteristic polynomial. Hence, we
must consider y(n) - y(n-1).
y(n-1) = 18 + y(n-2) + y(n-3)

=> y(n) - y(n-1) = y(n-1) - y(n-3)
=> y(n) - 2y(n-2) + y(n-3) = 0

From the above difference equation, we may find the characteristic equation:
r^n - 2r^(n-1) + r^(n-3) = 0

After factoring r^(n-3) on the left side, we get...
r^(n-3) * (r^3 - 2r^2 + 1) = 0

We must find the roots of the polynomial (r^3 - 2r^2 + 1) = 0.
By simple observation, we find that one of the roots is r = 1. This is because 1^3 - 2(1^2) + 1 = 0. Knowing this root, we can
factor the original polynomial by (r - 1) to get a quadratic term that will allow us to utilize the quadratic formula. Hence,
we divide (r^3 - 2r^2 + 1) / (r - 1) = r^2 - r - 1.
Thus, we rewrite the original characteristic polynomial as: (r^3 - 2r^2 + 1) = (r - 1)(r^2 - r - 1).

Using the quadratic formula, we find that the roots of the polynomial term r^2 - r - 1 are (1 - sqrt(5) / 2) and (1 + sqrt(5)
/ 2). These roots evaluate to approximately -0.618, and 1.618, respectively.

Hence we have three roots for our cubic characteristic polynomial:
r1 = 1
r2 = -0.618
r3 = 1.618

We have 3 roots, and therefore N = 3.

Now that we have computed the roots of the characteristic equation, we can express the general form of the solution to the
```

```
difference equation.
The general form is as follows:
y(n) = sum of 1 to N : ((sum of 0 to (mi - 1) : (cj * n^j)) * ri^n), where each cj is a constant

Since the multiplicity (mi) of each of our roots is 1, we know that mi - 1 = 0 for all i.
By filling in N=3, we can express the general solution as...
y(n) = sum of 1 to 3 : ( ci * ri^n ), where each ci is a constant.
y(n) = (c1 * r1^n) + (c2 * r2^n) + (c3 * r3^n)

By filling in the root values r1, r2, and r3 and simplifying, we get...
y(n) = c1 + (c2 * -0.618^n) + (c3 * 1.618^n)

We have our general solution above. We must apply our initial conditions y(0)=3, y(1)=5, and y(2)=26 to find the coefficients
c1, c2, and c3.
y(0) = 3 = c1 + c2 + c3
y(1) = 5 = c1 + (c2 * -0.618) + (c3 * 1.618)
y(2) = 26 = c1 + c1 + (c2 * -0.618^2) + (c3 * 1.618^2)

Using a 3x3 matrix and inverting it, we solve this system of equations above to get the values of c1, c2, and c3. The values
are:
c1 = -18
c2 = ((21 - 5 * sqrt(5)) / 2) = 4.9098
c3 = ((21 + 5 * sqrt(5)) / 2) = 16.0902

Hence, we get the particular form of our solution as a function of only n:
y(n) = -18 + (4.9098 * -0.618^n) + (16.0902 * 1.618^n)
```

Problem 2.34

Assume the declaration for 'func' is "int func(int a, int b)".

```
# Lbl: f
addi $sp, $sp, -20      # Allocate stack frame (4 for each argument (4 args) + 4 for return address)
sw $ra, 16($sp)         # Store return address $ra at the top address of the stack frame
sw $a3, 12($sp)         # Push arguments $a0, $a1, $a2, $a3 in reverse order (right to left) onto stack
sw $a2, 8($sp)
sw $a1, 4($sp)
sw $a0, 0($sp)
jal func                # Since $a0 and $a1 are the same in our inner call to func, just jump and link func
add $a0, $v0, $zero     # move the return value from the inner call to func into the $a0 register (first arg)
add $a1, $a3, $a4       # the 2nd arg to the outer func call is c + d
jal func                # jump and link to func (outer func call) -> we want to return what this returns
lw $a0, 0($sp)          # Reload argument registers from the stack
lw $a1, 4($sp)
lw $a2, 8($sp)
lw $a3, 12($sp)
lw $ra, 16($sp)         # Reload the return address register from the stack
jr $ra                  # Since $v0 is already set from the last call to func, just return

# Lbl: func
...
```

Problem 2.46

2.46.1

```
Suppose a program under the original architecture has the following measures:

Na = # of arithmetic instructions
Nl = # of load/store instructions
Nb = # of branch instructions
```

```
N = Na + Nl + Nb = # of total program instructions

CPIa = cycles per arithmetic instruction
CPIl = cycles per load/store instruction
CPIb = cycles per branch instruction

Ca = Na * CPIa = # of cycles of arithmetic instructions
Cl = Nl * CPIl = # of cycles of load/store instructions
Cb = Nb * CPIb = # of cycles of branch instructions
C = Ca + Cl + Cb = # of total CPU cycles for program

T = time / 1 CPU cycle = clock cycle time

t = T * C = program time execution

Now, we consider a new architecture that provides the following:
- reduces # of arithmetic instructions by 25%
- clock cycle time increases by 10%

We can describe these changes as follows:
Na' = 0.75 * Na
=> Ca' = 0.75 * Ca
=> C' = C - (0.25 * Ca)
T' = 1.1 * T

We express the following:
t = T * C
t' = T' * C' = (1.1 * T) * (C - (0.25 * Ca))

s = t / t' = speedup of program from original architecture to new architecture

For some program, we want to compute the value for r.

From the problem statement, we know that the given program has the following parameters filled:
Na = 500,000,000
Nl = 300,000,000
Nb = 100,000,000
CPIa = 1
CPIl = 10
CPIb = 3

Hence, by plugging these values in to the equations, we find the speedup of the new architecture compared to the original
architecture:
s = 0.94

This indicates that the new architecture had a speedup factor of less than 1 for the given program, which is equivalent to a
slowdown.
Therefore, we conclude that this is not a good design choice for the given program.
```

2.46.2

```
Suppose that an improved architecture offers double performance on arithmetic instructions.
We want to compute the speedup effect this has on our target program.

CPIa' = 0.5 * CPIa
=> Ca' = 0.5 * Ca
=> C' = C - (0.5 * Ca)

We want to compute:
s = t / t' = (T * C) / (T * C') = C / C'

By plugging in the program parameters shown in part 1, we get a speedup of:
s = 1.0704

If the upgraded architecture could offer 10x performance on arithmetic instructions, then the speedup factor could be computed
as follows:
```

```
CPIa' = 0.1 * CPIa

Using the same logic as above, we get:
s = 1.1343

In conclusion,
If arithmetic instructions get 2x improvement: speedup = 1.0704
If arithmetic instructions get 10x improvement: speedup = 1.1343
```