CMU 18-447 Introduction to Computer Architecture, Spring 2013
HW 3: Microprogramming Wrap-up and Pipelining

Instructor: Prof. Onur Mutlu
TAs: Justin Meza, Yoongu Kim, Jason Lin

Assigned: Mon., 2/11, 2013
Due: **Mon., 2/25, 2013 (Midnight)**
Handin: `/afs/ece/class/ece447/handin/hw3`

1 Adding the `REP MOVSB` Instruction to the LC-3b [25 points]

Let's say you're the lead architect of the next flagship processor at Advanced Number Devices (AND). You've decided that you want to use the LC-3b ISA for your next product, the Flopteron, but your customers want a smaller semantic gap and marketing is on your case about it. So, you've decided to implement your favorite x86 instruction, `REP MOVSB`, in LC-3b.

Specifically, you want to implement the following definition for `REP MOVSB` (in LC-3b parlance):

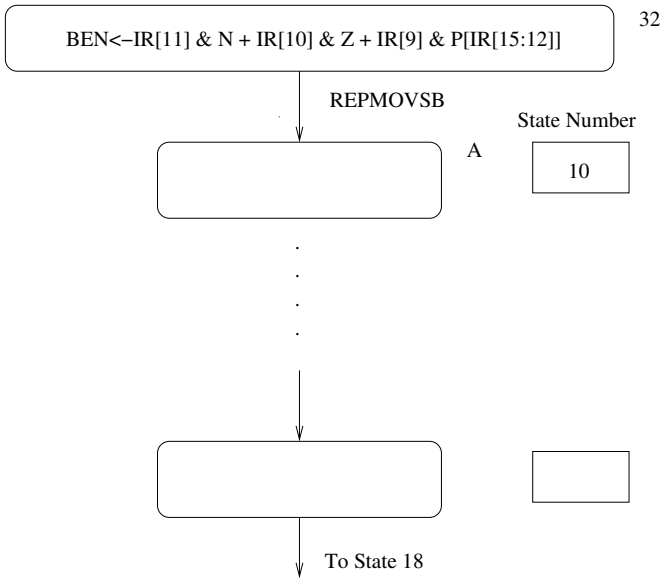
`REPMOVSB SR1, SR2, DR`

which is encoded in LC-3b machine code as:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1010				DR			SR1			0	0	0	SR2		

`REPMOVSB` uses three registers: `SR1` (count), `SR2` (source) and `DR` (destination). It moves a byte from memory at address `SR2` to memory at address `DR`, and then increments both pointers by one. This is repeated `SR1` times. Thus, the instruction copies `SR1` bytes from address `SR2` to address `DR`. Assume that the value in `SR1` is greater than or equal to zero.

(a) Complete the state diagram shown below, using the notation of the LC-3b state diagram. Describe inside each bubble what happens in each state and assign each state an appropriate state number. Add additional states not present in the original LC-3b design as you see fit.



(b) Add to the LC-3b datapath any additional structures and any additional control signals needed to implement `REPMOVSB`. Clearly label your additional control signals with descriptive names. Describe what value each control signal would take to control the datapath in a particular way.

(c) Describe any changes you need to make to the LC-3b microsequencer. Add any additional logic and control signals you need. Clearly describe the purpose and fuction of each signal and the values it would take to control the microsequencer in a particular way.

(d) The processing in each state is controlled by asserting or negating each control signal. Enter a 1 or a 0 or an X as appropriate for the microinstructions corresponding to the new states you added, in the `repmovsb.csv` file handed out with this homework. Add all your additional control signals at the end of the control word.

2 Pipelining [15 points]

Here is a program:

```
MUL R3  ← R1, R2
ADD R5  ← R4, R3
ADD R6  ← R4, R1
MUL R7  ← R8, R9
ADD R4  ← R3, R7
MUL R10 ← R5, R6
```

Calculate the number of cycles it takes to execute this program on the following machines:

(a) A non-pipelined machine

(b) A pipelined machine with scoreboarding and five adders and five multipliers without data forwarding

(c) A pipelined machine with scoreboarding and five adders and five multipliers with data forwarding

(d) A pipelined machine with scoreboarding and one adder and one multiplier without data forwarding

(e) A pipelined machine with scoreboarding and one adder and one multiplier with data forwarding

For all machine models, use the following 4 stages for the basic instruction cycle:

1. Fetch (1 clock cycle)
2. Decode (1 clock cycle)
3. Execute
   - `MUL` (5 clock cycles)
   - `ADD` (2 clock cycles)
   - The multiplier and adder are not pipelined
4. Write-back (2 clock cycles)

Please list any assumptions you make about the pipeline structure (for example, how data forwarding done between pipeline stages).

3 Delay Slots [30 points]

A machine has a five-stage pipeline consisting of fetch, decode, execute, memory access, and write-back stages. The machine uses delay slots to handle control dependences. Jump targets, branch targets, and destinations are resolved in the execute stage.

(a) What is the number of delay slots needed to ensure correct operation?

(b) Which instruction(s) in the assembly sequences below would you place in the delay slot(s), assuming the number of delay slots you answered for part (a)? Clearly rewrite the code with the appropriate instruction(s) in the delay slot(s).

(i)
```
ADD R5 ← R4, R3
OR  R3 ← R1, R2
SUB R7 ← R5, R6
J   X
[ Delay Slots ]
LW R10 ← (R7)
ADD R6 ← R1, R2
X:
```

(ii)
```
ADD R5 ← R4, R3
OR  R3 ← R1, R2
SUB R7 ← R5, R6
BEQ R5 ← R7, X
[ Delay Slots ]
LW R10 ← (R7)
ADD R6 ← R1, R2
X:
```

(iii)
```
ADD R2 ← R4, R3
OR  R5 ← R1, R2
SUB R7 ← R5, R6
BEQ R5 ← R7, X
[ Delay Slots ]
LW R10 ← (R7)
ADD R6 ← R1, R2
X:
```

(c) Can you modify the pipeline to reduce the number of delay slots (without introducing branch prediction)? Clearly state your solution and explain why.

4 Hardware vs. Software Interlocking [30 points]

Consider two pipelined machines, X and Y:

Machine *X* implements interlocking in hardware. On detection of a flow dependence, it stalls the instruction in the decode stage of the pipeline (blocking fetch/decode of subsequent instructions) until all of the instruction's sources are available. Assume internal register file forwarding (an instruction writes into a register in the first half of a cycle and another instruction can access the same register in the next half of the cycle). No other data forwarding is implemented. However, there are three execute units with adders, and independent instructions can be executed in separate execution units and written back out-of-order. There is one write-back stage per execute unit, so an instruction can write-back as soon as it finishes execution.

Machine *Y* does not implement interlocking in hardware. It assumes all instructions are independent and relies on the compiler to order instructions such that there is sufficient distance between dependent instructions. The compiler either moves other independent instructions between two dependent instructions, if it can find such instructions, or otherwise, inserts `NOP`s. Assume internal register file forwarding (an instruction writes into a register in the first half of a cycle and another instruction can access the same register in the next half of the cycle).

Both machines have 3 `ADD` execution units and the following 4 pipeline stages:

1. Fetch (1 clock cycle)

2. Decode (1 clock cycle)
3. Execute
   - `ADD` (3 clock cycles)
   - Each `ADD` execution unit is not pipelined, but an instruction can be executed if an unused `ADD` execution unit is available
4. Write-back (2 clock cycles)
   - There is one write-back stage per `ADD` execution unit

Consider the following 2 programs:

Program *A*

```
ADD R5  ← R6, R7
ADD R3  ← R5, R4
ADD R6  ← R3, R8
ADD R9  ← R6, R3
ADD R12 ← R3, R7
```

Program *B*

```
ADD R3  ← R1, R2
ADD R8  ← R9, R10
ADD R4  ← R5, R6
ADD R7  ← R1, R4
ADD R12 ← R8, R2
ADD R14 ← R8, R4
```

(a) Calculate the number of cycles it takes to execute each of these two code segments on Machines X and Machine Y.

(b) Which machine takes a smaller number of cycles to execute each Program A and Program B?

(c) Does the machine that takes the smaller number of cycles for Program A also take the smaller number of cycles than the other machine for Program B? Why or why not?

(d) Would you say that the machine that provides a smaller number of cycles as compared to the other machine has higher performance (taking into account all components of the performance equation we discussed in class)?

(e) Calculate the instruction code size of each of these two code segments on Machine X and Y, assuming a fixed-length ISA, where each instruction is encoded as 4 bytes.

(f) Which machine incurs lower instruction code size for Program A and Program B?

(g) Does the same machine incur lower instruction code size for both Program A and Program B? Why or why not?