CS-UY 2214 — Homework 6

Jeff Epstein

Introduction

Unless otherwise specified, put your answers in a plain text file named `hw6.txt`. Number each answer. Submit your work on Gradescope.

You may consult the E20 manual, which is available on Brightspace.

Problems

1. We previously developed a way to perform integer multiplication in software on E15. The approach we used, when calculating $m \times n$, was to add m to itself, n times in a loop. This approach gives us the right answer, but is terribly inefficient, operating in linear time proportional to the magnitude of n. For large numbers this can be prohibitively slow.

   The E20 has a more expressive machine language, so we are able to implement a more elegant multiplication algorithm. Before we discuss that, let's review the standard elementary school approach to multiplication of decimal numbers. For example, let's consider the case of calculating the product 175×21:

$$\begin{array}{ccccc} & & 1 & 7 & 5 \\ \times & & & 2 & 1 \\ \hline & ? & ? & ? & ? \end{array}$$

   The solution to the above problem can be understood as follows:

$$\begin{array}{cccccc} & 5 & \times & 1 & \times & 21 \\ + & 7 & \times & 10 & \times & 21 \\ + & 1 & \times & 100 & \times & 21 \\ \hline = & & 3675 & & & \end{array}$$

   In other words, we're going to multiply the second factor by *each digit* of the first factor, multiplied by its position in the decimal number. That is, 5 is in the one's place, 7 is the 10's place, and 1 is in the 100's place.

   The same algorithm works in binary. Thus, the following product:

$$\begin{array}{ccccccc} & 1 & 0 & 1 & 0 & 1 & 1 \\ \times & & & & 1 & 0 & 1 \\ \hline & ? & ? & ? & ? & ? & ? \end{array}$$

   Can be understood as follows:

$$\begin{array}{cccccc}
 & 1 & \times & 1_2 & \times & 101_2 \\
+ & 1 & \times & 10_2 & \times & 101_2 \\
+ & 0 & \times & 100_2 & \times & 101_2 \\
+ & 1 & \times & 1000_2 & \times & 101_2 \\
+ & 0 & \times & 10000_2 & \times & 101_2 \\
+ & 1 & \times & 100000_2 & \times & 101_2 \\
\hline
= & \multicolumn{3}{c}{11010111_2} & &
\end{array}$$

The naive algorithm (wherein we add m to itself, n times in a loop) has a runtime proportional to the magnitude of n. If n is large, this can result in unreasonable performance. The algorithm described here (wherein we sum the product of m times each bit of n), which you are required to use in your solution, has a runtime proportional to the number of *bits* in n, i.e. $O(log_2 n)$.

Your task is to use the algorithm described above to implement an efficient multiply program in E20 assembly language. Your program must include labels named `multiplicand` and `multiplier`, referring to values in memory that are to be multiplied. Initially set these values to 123 and 119. Therefore, your code should include the following lines:

```
multiplicand: .fill 123
multiplier: .fill 119
```

Your program must read these values from memory and perform the algorithm on them, leaving the product in register `$1` before halting. You may assume that the two input numbers are at most 7 bits wide and that both are non-negative; this ensures that their (non-negative) result fits into a 16-bit register. You must implement the algorithm described above. Your program must use a loop to successively add each value: do not perform each addition with several separate sections of code.

Your solution must be thoroughly commented, or else it will not be graded. Make sure that the comments help the grader understand the intent of your code. Test your solution using your assembler and simulator.

Put your answer in a file named `mul.s`.

Hints:

- I recommend using a bitmask to successively examine each bit in n.
- Recall that you can perform a binary left shift by adding a number to itself. For example `0100 == 0010 + 0010`.
- Given a bitmask with a single bit on, you can determine if that bit is on in another number by ANDing them together and determining if the result is zero.

2. Consider the following complete E20 program:

```
    lw $1, first($0)
    lw $2, second($0)
    add $3, $1, $2

first:
    .fill 8618
second:
    .fill 16388
```

Without entering this code into your computer, determine what is the final value of register `$3`. Justify your answer.

3. Consider the E20 assembly program shown below.

```
    add $1, $0, $0
    add $4, $0, $0
    lw $3, data($0)
Loop:
    slti $1, $3, 18
    jeq $1, $0, Skip
    add $4, $4, $3
    addi $3, $3, 1
    jeq $0, $0, Loop
Skip:
    halt
data:
    .fill 16
```

Assume that the E20 processor has an average CPI of 4, and that the processor's clock is operating at 1 MHz (one million cycles per second).

(a) How many instructions will be executed before the program halts? Do not count the `halt` instruction itself.

(b) How long will the program take to execute? Give your answer in clock cycles.

(c) How long will the program take to execute? Give your answer in microseconds (μs).

4. Consider the following E20 program:

```
    movi $1, 0           # instr a
    lw $2, grade1($0)    # instr b
    add $1, $1, $2       # instr c
    lw $3, grade2($0)    # instr d
    add $1, $0, $3       # instr e
    halt

grade1:
    .fill 72
grade2:
    .fill 99
```

Assume that the program is executed on a 5-stage pipelined E20 processor, as we discussed in class. Assume that every instruction uses every stage.

(a) Identify all read-after-write data conflicts. For each data conflict, indicate the instructions that conflict (identified by their letter, shown in comments), and indicate which register or memory location is in contention.

(b) Complete the following table, showing the step-by-step execution of the program in the pipeline. Resolve hazards with stalls (i.e. by inserting bubbles/`nop`s as necessary). Assume no forwarding. Assume that the MEM stage completes in one cycle. At each cell in the table, write the letter of the instruction that will execute at that stage, or leave the cell empty in case of a bubble. Use as many rows as necessary to show the complete execution of each lettered instruction. The first row has been completed for you. Don't include the `halt` instruction in your table.

```
Time    |  IF    |   ID   |   EX   |  MEM   |  WB
 1      |   a    |        |        |        |
 2      |        |        |        |        |
 3      |        |        |        |        |
 4      |        |        |        |        |
 ....
```

(c) Re-order the lettered instructions in the given program to reduce the necessary stalls. Give your answer as an ordered sequence of the letters $a \ldots e$, where each letter corresponds to one of the instructions in the program. The goal is to re-order the instructions so that the program can execute as fast as possible.

Your re-ordered program must produce identical results as the original program. To ensure that, make sure that you don't re-order any instructions that conflict with each other. For example, let's suppose that we have some code with three instructions: `x`, `y`, and `z`, in that order. Further let's assume that `x` has a read-after-write conflict with `y`, and that `x` also has a read-after-write conflict with `z`. There is no conflict between `y` and `z`. We can therefore re-order the code as `xzy`, but not as `zxy`.

(d) Complete an execution table again, under the same rules as before. However, this time show the execution of your program with re-ordered instructions, as you answered in the previous part.

```
Time    |   IF   |   ID   |   EX   |  MEM   |  WB
 1      |        |        |        |        |
 2      |        |        |        |        |
 3      |        |        |        |        |
 4      |        |        |        |        |
 ....
```

5. The XQ1 processor has a 5-stage pipeline. The following table shows the execution time of each stage of the pipeline, measured in picoseconds (ps), or trillionths of a second.

Stage	IF	ID	EX	MEM	WB
Time	180ps	150ps	400ps	230ps	100ps

(a) Based on the table above, what is the maximum clock frequency of the XQ1 processor? Give your answer in gigahertz (GHz).

(b) The XQ2 processor attempts to rectify a shortcoming of its predecessor, the XQ1, namely, its unbalanced pipeline stages. To achieve this, the XQ2 splits the EX stage into two parts, resulting in a 6-stage pipeline. The following table shows the execution time of each stage of the pipeline in the XQ2 processor.

Stage	IF	ID	EX1	EX2	MEM	WB
Time	180ps	150ps	250ps	150ps	230ps	100ps

Based on the table above, what is the maximum clock frequency of the XQ2 processor? Give your answer in gigahertz (GHz).

6. Consider the following E20 assembly language program, which will find the largest number in an array. You've seen this program before.

```
main:
    movi $1, 0           # ram [0]
    movi $7, 0           # ram [1]
repeat:
    lw $2, array ($1)    # ram [2]
    jeq $2, $0, done      # ram [3]
    slt $5, $7, $2       # ram [4]
    jeq $5, $0, next      # ram [5]
    add $7, $0, $2       # ram [6]
next:
    addi $1, $1, 1       # ram [7]
    j repeat             # ram [8]
done:
    halt                 # ram [9]
array:
    .fill 53             # ram [10]
    .fill 22             # ram [11]
    .fill 94             # ram [12]
    .fill 2              # ram [13]
    .fill 19             # ram [14]
    .fill 0              # ram [15]
```

A conditional jump in the form of a `jeq` instruction can cause a control hazard because it's not clear which instruction should be put in the pipeline next. Each conditional jump may have one of two possible resolutions: either it will *branch* (i.e. the next instruction in the pipeline is the target of the jump) or it will not *not branch* (i.e. the next instruction in the pipeline is the instruction in the subsequent memory location after the conditional jump instruction). Each `jeq` instruction has an independent predictor bit.

In all cases, if the next instruction is predicted wrong (a *misprediction*), it must be squashed from the pipeline, and the correct instruction fetched after a bubble. To avoid the penalty of frequent mispredictions, we want to choose the best possible *branch predictor*.

In case of control hazards caused by a conditional jump instruction such as `jeq`, stalls can be reduced by using a branch predictor. In class, we discussed three kinds of branch predictors:

- Predict branch taken — After a conditional jump, the next instruction in the pipeline will the be the target of the jump.
- Predict branch not taken — After a conditional jump, the next instruction in the pipeline will be the instruction at the subsequent memory address.
- Dynamic prediction — After a conditional jump, the next instruction will be chosen based on the resolution of the previous execution of that particular conditional jump. If there was no previous execution of that conditional jump, predict that the branch will not be taken.

In answering the following questions about the above E20 program, assume that each misprediction results in a penalty of 3 clock cycles, occupied by a pipeline bubble while the correct instruction is executed. For the purposes of this exercise, you can ignore other hazards.

(a) What will be the total misprediction penalty accrued in the above program, if it is run on a processor that always predicts branch taken? Justify your answer.

(b) What will be the total misprediction penalty accrued in the above program, if it is run on a processor that always predicts branch *not* taken? Justify your answer.

(c) What will be the total misprediction penalty accrued in the above program, if it is run on a processor that uses dynamic prediction based on the previous resolution of that conditional jump? Justify your answer.