UNIVERSITY OF CALIFORNIA, DAVIS
Department of Electrical and Computer Engineering

EEC171 Parallel Computer Architectures Spring 2017

Homework 1 Due Thursday 20 April 2017 10:00 am in homework box **NOT IN CLASS**

As a professor, I want to give you interesting homework problems and make sure that you cover the breadth of the course material. This is a 4 unit class and I'm only giving 3 homework assignments, so I feel justified in giving homework assignments that are fairly involved. However, I know that your time is valuable, so we'll try allowing you to select from a range of problems. Please let me know what you think about this policy and also about what you think about the amount of homework.

The homework is to choose 6 of 8 book problems and to do at least half the point-value of problems from the written questions below (this also gives you some visibility into what my exams look like). I (well, Terry) will distribute solutions to all problems in class on Thursday 20 April.

Problems from H&P 5th ed. Please do 6 of the following 8 problems.

- 3.1–3.6. Explores instruction scheduling, reordering, multiple-issue.
- 3.10. VLIW & register use.
- 3.11. Branch prediction.

Please note that while these problems are good problems, they are not as clearly written as you might like. When Bakos/Colwell use the term "latencies beyond single cycle", what they mean is that every instruction takes at least one cycle and `+n` means `n` additional cycles (so `LD` takes 4 cycles overall). The branch delay slot and the `+1` for branches just means "branches take two cycles to finish no matter what" (all of their solutions basically assume there's an empty cycle at the end of every piece of code because of the branch; he never fills the branch delay slot; I don't necessarily agree with this, but for the purposes of these problems, let's go with what they used in their solutions). Please ask for assistance if things aren't clear.

For clarity, here I'm copying some of the explanations from the answers, without the answers:

3.1 "Each instruction requires one clock cycle of execution (a clock cycle in which that instruction, and only that instruction, is occupying the execution units; since every instruction must execute, the loop will take at least that many clock cycles). To that base number, we add the extra latency cycles. DonâĂŹt forget the branch shadow cycle."

3.2 "Remember, the point of the extra latency cycles is to allow an instruction to complete whatever actions it needs, in order to produce its correct output. Until that output is ready, no dependent instructions can be executed."

3.11 "The convention is that an instruction does not enter the execution phase until all of its operands are ready."

3.13 and 3.14 are terrific problems, and I'm really tempted to assign them, but out of respect for your time, probably not a good idea. However, you should look through them—I'd expect you to be able to do a problem like that.

Additional questions. This is the ILP midterm in 2013 and is thus great preparation for an exam. There are 60 points of questions below. You must do at least 30 points of questions for this homework, although all the questions are good practice. You would have had an entire class (110 minutes) to complete this exam.

1. The ARMv7-M instruction set has a feature called "conditional execution" for instructions. The ARM manual defines it as follows:

   > *Conditionally executed* means that the instruction only has its normal effect on the programmers' model operation, memory and coprocessors if the N, Z, C and V flags in the APSR [Application Program Status Register] satisfy a condition specified in the instruction. If the flags do not satisfy this condition, the instruction acts as a NOP, that is, execution advances to the next instruction as normal, including any relevant checks for exceptions being taken, but has no other effect.

   Let me paraphrase. When an ARM implementation runs the instruction `CMP`, that sets a set of flags. Those flags together encode particular results of the comparison, such as equality or less-than. For the purposes of this question, those flags remain set until another `CMP` instruction is executed. Future instructions can be "conditionally executed" based on the flags set by `CMP`: the programmer can designate a conditionally-executed instruction by appending a condition (such as `EQ` for equality or `LT` for less-than) to the opcode for that instruction. The conditionally-executed instruction only runs if the flags satisfy the designated condition. (Of course, if no condition is specified, the instruction runs unconditionally.) In ARM, branch instructions also use the result of `CMP` instructions (as opposed to MIPS, where `BEQ` both executes the comparison and branches based on it).

   As an example, consider the following code sequence that checks if `r3` is zero and if not, performs an add operation.

```
      CMP r3, #0     // compare r3 to zero
      BEQ skip       // if the previous CMP indicated equality,
                     // jump to "skip"
      ADD r0, r1, r2 // r0 = r1 + r2
skip: ...
```

   A clever ARM programmer can rewrite this using conditional execution:

```
      CMP r3, #0       // compare r3 to zero
      ADDNE r0, r1, r2 // only if the previous comparison was
                       // not-equal, compute r0 = r1 + r2
```

   Note that `ADDNE` is an `ADD` instruction with the condition `NE` (not equal), where the condition was set on the previous `CMP` (comparison) operation. Other conditions that you may find useful: `EQ` (equal), `GT` (greater-than), and `LE` (less-than-or-equal).

(a) (2 points) In one word, **what is the concept we discussed in class** that describes what we're doing when we conditionally execute an ARM instruction?

(b) (4 points) **Name two advantages of adding conditional execution to an instruction set.**

(c) (2 points) **Name one potential disadvantage of adding conditional execution to an instruction set.**

(d) (5 points) **Explain** (in English, completely but as briefly as possible) **what the following ARM code does.** You should need no more than one sentence.

```
CMP   r0, #0
CMPNE r0, #1
MOVEQ r1, #1
```

(Notes: `#` indicates an immediate value [a constant]; `MOV` copies its second argument to its first argument; `EQ` indicates conditional execution if the previous `CMP` [comparison] operation indicated the first argument is equal to the second; `NE` indicates conditional execution if the previous `CMP` [comparison] operation indicated the first argument is not equal to the second.)

(e) (5 points) Using the operations `CMP` and `MOV`, and using conditional operations as appropriate, **write a segment of ARM code as succinctly as possible that implements the following**:

If `r0` is less than or equal to zero, set it to 0, otherwise set it to 1.

2. In image processing, the Prewitt operator is used for edge detection. You can skip all the math below if you don't care; you don't need the math to solve the problem.

   If we define A as the source image, and $*$ the convolution operator,

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} * A; \qquad G_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} * A,$$

   and

$$G = \sqrt{G_x^2 + G_y^2}; \qquad \Theta = \arctan(G_y, G_x).$$

   The higher the gradient G, the more likely the pixel represents an edge; Θ represents the direction of that edge. If you wanted to skip all the math above, you can start reading again now. And if I got any math wrong, use my incorrect math. In this problem, we will compute G (the magnitude of the gradient) and Θ (the direction of the gradient) for one output pixel in the source image. For the purposes of this problem, assume all arithmetic operations are both commutative and associative.

   The Prewitt operator requires the input of eight input pixels, which we will designate as `A1`–`A9`, skipping `A5` (since `A5` is unused). Both the eight input pixels and two output values will be in main memory.

   In assembly, this code follows. Assume each line of code is a real hardware instruction (even the weird `arctan` instruction). There are 25 instructions in the program.

```
01 r1 = Mem[Ma1]  # load
02 r2 = Mem[Ma2]  # load
03 r3 = Mem[Ma3]  # load
04 r4 = Mem[Ma4]  # load
05 r6 = Mem[Ma6]  # load
06 r7 = Mem[Ma7]  # load
07 r8 = Mem[Ma8]  # load
08 r9 = Mem[Ma9]  # load

09 rx = r3 - r1
10 rx = rx - r4
11 rx = rx + r6
12 rx = rx - r7
13 rx = rx + r9

14 ry = r1 - r7
15 ry = ry + r2
16 ry = ry - r8
17 ry = ry + r3
18 ry = ry - r9

19 rtheta = arctan(ry, rx)

20 rx = rx * rx
21 ry = ry * ry
22 rg = rx + ry
23 rg = sqrt(rg)

24 Mem[Mrg] = rg         # store
25 Mem[Mtheta] = rtheta  # store
```

On all parts of this problem, unless told otherwise, assume a perfect machine: infinite renaming registers and instruction windows, out-of-order issue/execution/commit, perfect memory disambiguation and branch prediction, infinitely wide issue, single-cycle {fetch/decode/execute/commit} for any instruction (any instruction begins and ends in the same cycle), etc.

You may wish to draw a dependency graph to help you, and we recommend you do it below. Be very careful to draw this graph correctly, because if you do it wrong, it'll affect all your answers; but we'll try to grade accordingly if you do it incorrectly. But do it right.

(a) (4 points) Measured in instructions per cycle, **what is the instruction-level parallelism of this code segment?**

(b) (6 points) Assume you want to build a VLIW processor that will run this code segment in such a way that it achieves the maximum instruction-level parallelism. Available to you are several kinds of functional units: `+/-` (can run addition and subtraction instructions); `mem` (can run load and store instructions); `*` (can run multiply instructions); and `sp` (special) (can run `arctan` and `sqrt` instructions).

What is the minimum mix of functional units (for instance, "8 `+/-`, 4 `mem`, and 1 `*`") **that you must build into your VLIW processor to allow this code segment to run in such a way that it achieves the maximum instruction-level parallelism?**

(c) (4 points) Your supervisor accepts your design from the previous part, but then returns to you with the design, indicating that the chuckleheads in charge of corporate finance mandate that you must cut costs by eliminating one functional unit from your design with minimal performance loss and while still maintaining correct computation. (To maintain correct computation, you still require at least one instance of each kind of functional unit.) **Which unit do you eliminate (explain your reasoning please) and how much slower will this code segment run as a result?**

(d) (5 points) Your colleague Ben Bitdiddle thinks he can make your code run in the same number of instructions but *fewer cycles* by arranging the computation differently and adding more functional units to your VLIW design. (In other words, rewrite parts of the code to expose more instruction-level parallelism.) In English, **describe how you would change the code segment** (how you would write different assembly code) **with the same result, and quantify how rewriting the code reduces the number of**

cycles to run this code segment. You may use diagrams (or new assembly code) to support your answer.

3. Consider a machine with the following characteristics: the machine has the ability to issue multiple operations in a single cycle, all operations take a single cycle to {fetch-decode-execute-commit}, and you may not schedule two instructions on the same cycle if they both access the same register and at least one of those accesses is a write. (Two reads is OK.)

   Now consider the following code segment, which has at least one of each of {RAW, WAW, WAR} hazards.

```
add r3,r2,r1 # r3 = r2 + r1
sub r2,r4,r1 # r2 = r4 - r1
mul r5,r2,r1 # r5 = r2 * r1
div r5,r3,r4 # r5 = r3 / r4
```

   For each type of hazard, identify (1) an instance of the hazard in the above code segment and (2) how register renaming for that hazard would potentially help performance.

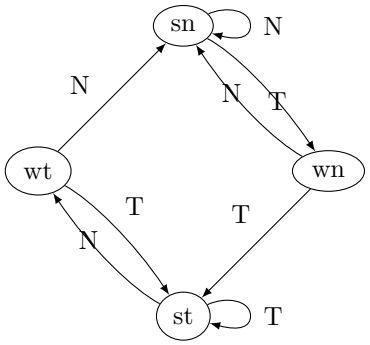
   "An instance of the hazard in the above code segment" should be expressed as "`ry` between instruction `op1` and instruction `op2`".

   (a) RAW (read-after-write) hazard
      i. (2 points) Identify an instance of this hazard in the above code segment.
      ii. (2 points) How would register renaming for this hazard potentially help performance? If register renaming would not help, say "Register renaming will not help".

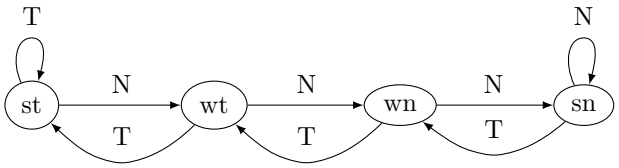
   (b) WAW (write-after-write) hazard
      i. (2 points) Identify an instance of this hazard in the above code segment.
      ii. (2 points) How would register renaming for this hazard potentially help performance? If register renaming would not help, say "Register renaming will not help".

   (c) WAR (write-after-read) hazard
      i. (2 points) Identify an instance of this hazard in the above code segment.
      ii. (2 points) How would register renaming for this hazard potentially help performance? If register renaming would not help, say "Register renaming will not help".

4. (8 points) In class we noted that the 2-bit branch predictor I presented was different than the traditional formulation of a 2-bit branch predictor. In the below state diagrams, "st" is "strongly taken"; "wt" is "weakly taken"; "wn" is "weakly not taken"; and "sn" is "strongly not taken". "st" and "wt" both predict the next branch will be taken; "wn" and "sn" predict not taken. When either predictor sees a new branch, it updates its state, depending on if the new branch was taken ("T") or not taken ("N").



Class predictor



Traditional predictor

Assume both predictors start in the `wn` state. **Which of two predictors would you choose for your hardware if your workload had an equal number of the following 3 branch patterns, and why?** Assume those patterns run for a very long time (you can ignore startup effects).

- `TNTNTNTN...` (alternating with period of 2)
- `TTNNTTNN...` (alternating with period of 4)
- `TTTNNNTTTNNN...` (alternating with period of 6)