Computer Architecture Assignment 4

Rachel Hwang

November 12, 2013

Book exercises

- 14.12.1a
  For a non-pipelined processor, the clock-cycle time is just the sum of all the stages, 250+350+150+300+200 = 1250ps.
  For a pipelined processor, it is just the length of the longest stage, 350ps.

- 14.12.2a
  For a non-piplined processor, the latency is just the clock-cycle time = 1250ps.
  For a pipelined processor, it is the number of stages times the longest stage = $350ps \cdot 5 = 1750$.

- 14.12.3a
  I would split the ID stage, reducing clock cycle time to 300ps.

- 14.12.4a
  The utilization of D-Mem = 35%.

- 14.12.5a
  The utilization of the write-register portion of Registers = 65%.

- 14.12.6a
  Assume we execute 100 instructions with the given distribution of ALU, BEQ, LW and SW instructions.
  Single-cycle organization takes $1250ps \cdot 100inst = 125000$ seconds.
  Multi-cycle organization has a clock rate of the longest stage, but only executes the stages necessary in each instruction type, so it will take $45inst \cdot (350ps \cdot 4stages) + 20inst \cdot (350ps \cdot 3stages) + 20inst \cdot (350ps \cdot 5stages) + 15inst \cdot (350ps \cdot 4stages) = 140000$ seconds.
  A pipelined organization has a clock rate of the longest stage, and five stages per instruction, but can execute five different instructions at once, so it will take $350ps \cdot 5stages \cdot 100inst \cdot .2 = 35000$ seconds.

  The multi-cycle organization will take about 1.12 times as long as the single-cycle.
  The pipelined organization will take about .28 times as long as the single-cycle.

- 14.15.1a
  First of all, this instruction is incompatible with the MIPS ISA because it has four operands (Rd, Rt, Rs *and* Offs) where MIPS encoding expects three. We would have to add to the MIPS encoding procedure, perhaps by adding another instruction type with a new OP code that encodes the Offs value to the shift-amount and FUNC fields (bits 10-0).

  As for the pipelined datapath, we would need to include another adder after the MEM component in order to add Rt to the value of Mem[Offs+Rs]. This adder would take Rt from the Registers (or 0 in other instruction types) and the output of the MEM read of Offs+Rs as inputs. To determine whether the input of this adder is 0 or Rt, we require another Mux, call it addSrc. We will also require a Mux in the circuit just before the sign extender, since we need to determine whether the sign extender (whose result will be fed into the ALUSrc Mux) will take 11 bits (in this instruction type) or 15 bits (in all other instruction types). Call that Mux ExtSrc.

- 14.15.2a
  The new control signals we will need are for the new Muxes addSrc and ExtSrc. The control will set addSrc to 1 for the new instruction type, indicating that Rt should be added to the output coming from MEM, or else 0 indicating that 0 should be added. The control will set ExtSrc to 1 in this instruction type, indicating that the sign extender should use only bits 10-0 as input, or else set 0 for the usual 15 bits.

- 14.15.3a
  This makes an existing hazard worse since it adds an additional stage to the datapath. For instance, in a load word instruction followed by an instruction that uses the result of that load as an operand, there is ordinarily a stall of two stages (two bubbles). In our new datapath, there will instead be a stall of three stages. Since no other instruction type will need the new adder step, anytime there would ordinarily be a stall as a following instruction waits for a preceeding instruction to finish traveling the data path, in our modified path, the stall is one stage longer. [This all assumes no forwarding.]

- 14.15.4a
  This instruction type is useful anywhere we want to use the value of a word from memory as an operand in addition. It would replace MIPS sequences of this format:

```
    LW   $t0, 4($t1)
    ADD  $s0, $t0, $s1

Replaced with:

    ADDM $s0, $s1+4($t1)
```

- 14.15.5a
  If this instruction already exists in some legacy ISA, a modern processor would likely just translate the CISC instruction that use it into the RISC instruction in that ISA.

- 14.15.6a
  Given that we replace 2 instructions with our new instruction every 30, we have 29 instructions in 29+2 cycles, which means the new CPI = $31/29 \approx 1.07$. This means that our new pipeline organization is actually *slower*, taking 1.07 times as long as the original.

- 14.24.1b
  Always taken: 60%. Always not taken: 40%.

- 14.24.2b
  It is 25% accurate.

- 14.24.3b
  It would be 60% accurate if the pattern repeated forever.

Lab Component

- 2.3.1
  The basic configuration of the atom_core is

```
 L1_I_0:
    type: cache
    size: 131072
    sets: 256
    ways: 8
    line_size: 64
    latency: 2
    pending_queue_size: 128
```

```
    config: writeback
  L1_D_0:
    type: cache
    size: 131072
    sets: 256
    ways: 8
    line_size: 64
    latency: 2
    pending_queue_size: 128
    config: writeback
  L2_0:
    type: cache
    size: 2097152
    sets: 4096
    ways: 8
    line_size: 64
    latency: 5
    pending_queue_size: 128
    config: writeback
  MEM_0:
    type: dram_cont
    RAM_size: 536870912
    number_of_banks: 64
    latency: 170
    latency_ns: 49.9853
    pending_queue_size: 128
  p2p_core_L1_I_0:
    type: interconnect
    latency: 0
  p2p_core_L1_D_0:
    type: interconnect
    latency: 0
  p2p_L1_I_0_L2_00:
    type: interconnect
    latency: 0
  p2p_L2_0_L1_D_00:
    type: interconnect
    latency: 0
  p2p_L2_0_MEM_00:
    type: interconnect
    latency: 0
```

- 2.3.2
  loop770000 takes 5261000 cycles to complete.
  Using the listed frequency of 1356664 Hz, loop770000 takes about 5261000/1356664 $\approx$ 3.88 seconds to complete.

- 2.4
  For loop770000, the atom processor fetches 2310273 instructions.
  The opclass distribution is:

```
logic: 83
addsub: 1540164
addsubc: 0
addshift: 0
sel: 1
cmp: 0
br.cc: 770022
jmp: 12
bru: 26
assist: 0
mf: 2
```

```
ld: 1540090
st: 770089
ld.pre: 0
shiftsimple: 2
shift: 0
mul: 0
bitscan: 0
flags: 40
chk: 8
fpu: 0
fp-div-sqrt: 0
fp-cmp: 0
fp-perm: 0
fp-cvt-i2f: 0
fp-cvt-f2i: 0
fp-cvt-f2f: 0
vec: 0
special: 0
```

loop770000 takes 770089 branch instructions.

- 2.5
  At 770056 instructions, alu0 completes most of the computation.

- 2.6
  For loop77, the atom core fetches 465 instructions.
  Comparing this result with the number for loop770000, $2310273/465 \neq 10000$, not even close. The ratio is $4968 : 1 \approx 5000 : 1$.

  However, the emptyone program which is empty still has 202 instructions fetched. This implies that there is a base of about 200 instructions fetched per program regardless of content. Subtracting this base number of instructions from the counts for loop77 and loop77000 gives a much more intuitive ratio of $2310071/263 \approx 8783 : 1$.

- 3.2
  According to matrix.st, matrix_multiplication takes 4208174 cycles to complete.

  In the commit stage, 1630814 instructions are committed.

  Using these numbers, the IPC = $1630814/4208174 \approx 0.375$, which is the same as the one listed in the commit section.

- 3.3
  In the fetch stage, 1663645 instructions are fetched.

  This is different from the number of icache accesses, 404290, because in x86, instruction size is variable, and so each icache access may get several instructions instead of just one.

  The average ratio of instructions fetched to icache accesses = $1663645/404290 \approx 4.11$.

- 3.4.1
  Using the single_core, matrix_multiplication takes 2657586 cycles to complete.

- 3.4.2
  In the commit stage, 1630813 instructions are committed.
  Yes, this is about the same as the number in the atom_core.

- 3.4.3
  Using the above numbers,
  single_core CPI = $2657586/1630813 \approx 1.63$
  atom_core CPI = $4208174/1630814 \approx 2.58$

  The number of instructions that a machine can execute at one time is more dependent on processor organization (ie. pipelined vs. not) than it is on CPI.

Exectution time with atom_core = $\frac{4208174 cycles}{2173160 cps} \approx 1.936$ seconds
Execution time with single_core = $\frac{2657586 cycles}{915451 cps} \approx 2.903$ seconds

So the atom_core execution time is about .667 times that of the single_core execution time.

If the two machines had the same clock rate, the single_core machine would be about 1.58 times faster than the atom_core.

- 3.4.4
  Atom_core pipeline:
  fetch → decode → issue → execution → commit
  Single_core pipeline:
  decode → dispatch → issue → execution → commit → fetch

- 3.4.5
  Single_core u-ops = 2221573
  Atom_core u-ops = 2221574

  Both machines execute about the same number of u-ops. This means that despite using different processor organizations, ultimately they translate their x86 instructions into the same simple u-ops.

- 3.4.6
  The fraction of decodes which are "fast":
  Single_core = $\frac{242}{254} \approx 95\%$

  Atom_core = $\frac{227}{232} \approx 97.8\%$

  So the fraction of "fast" decodes is pretty similar for both architectures.

- 3.4.7
  Yes, according to the stats file, the single_core execution requires 1 microcode assist.

- 3.4.8
  rob_reads: 6687263
  rob_writes: 2341958
  rename_table_reads: 2221573
  rename_table_writes: 1645382
  rob + rename_table reads: 8908836
  physreg_reads: 9043946 + 29 + 134524 + 68948 = 9247447
  rob + rename_table writes: 3987340
  physreg_writes: 4164472 + 84 + 279111 + 143029 = 4586696

  percentage of total reads from physreg_reads = $\frac{9247447}{8908836+9247447} \approx 50.93\%$
  percetage of total writes from physreg_writes = $\frac{4586696}{4586696+3987340} \approx 53.5\%$

  A simple single-cycle execution machine does not use forwarding circuits, therefore 100% of reads and writes would come from the physical registers.

- 3.4.9
  Single_core prediction success rate: = $\frac{68991}{70641} \approx 97.66\%$
  Atom_core prediction success rate: = $\frac{67401}{69053} \approx 97.61\%$

  Because the branch prediction success rate is so high for both machines, this means that the executes in one long stream with infrequent backtracking. This means that the program looks more like one long stream of instructions.

The average number of instructions between mispredicted branches is $\frac{1630814}{1652} \approx \frac{1630813}{1650} \approx 987$ instructions.

- 3.5.3
  These stats are from clustalw.stat.
  Total instruction count: 117952544
  Clock cycles run: 85716997
  Branch instructions: 11684618
  Vector instructions: 5173002
  Load instructions: 32408262
  Store instructions: 10859068