Christopher Daffron CS530 HW#1 9/5/14

1-1.13) a) If choosing based only on the overall SPEC performance, I would choose the Itanium 2, because its overall ~~SPEC~~ ratio of 27.12 is higher than the Opteron's SPEC ratio of 20.86

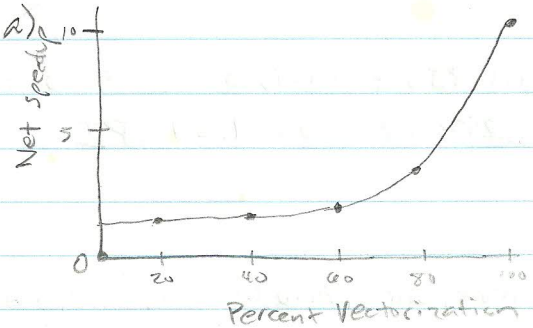
b) 60% wupwise, 20% ammp, 20% apsi

$(.6)(0.92) + (0.20)(1.03) + (0.20)(0.65) = 0.888 \Leftarrow \frac{\text{opteron time}}{\text{itanium time}}$

c) speedup = $\frac{\text{itanium time}}{\text{opteron time}} = \frac{1}{\text{opteron time}/\text{itanium time}} = \frac{1}{0.888} = 1.13$

2-1.14)

a)



$\text{Speedup} = \frac{1}{(1 - \text{Frac}_{\text{Enh}}) + \frac{\text{Frac}_{\text{Enh}}}{\text{speedup}}}$

b) $2 = \frac{1}{(1-F) + \frac{F}{10}}$ $F = 0.555$ 55.5% vectorization is needed for a speedup of 2

c) Proportion of computation time = $\frac{(0.555)(1/10)}{(0.555)(1/10) + (1-0.555)(1)} = \boxed{0.1109}$

d) $5 = \frac{1}{(1-F) + \frac{F}{10}}$ $F = 0.888$ 88.8% vectorization is required for half of max speedup

e) with 20 x speedup and 70% vectorization, speedup = $\frac{1}{(1-0.7) + \frac{0.7}{20}} = 2.985$

to get 2.985 overall speedup with 10 x vector speedup, they need

$2.985 = \frac{1}{(1-F) + \frac{F}{10}}$ $F = 0.739$, so need a 73.9% vectorization

3-1.15)

a) Eq from c above, $0.5 = \frac{(x)(1/10)}{(x)(1/10) + (1-x)(1)}$ $X = 0.909$

speedup = $\frac{1}{(1-0.909) + \frac{.909}{10}} = \boxed{5.5}$

b) As equated in part a, 90.9% of the original execution time has been converted to fast mode.

4-A.1) Assuming the instructions fit into the follow types:

ALU: add, sub, mul, compare, load imm, cond move, shift, AND, OR, XOR, other

Load/Store: load, store,

Conditional Branch: cond branch

Jump: jump, call, return

ALU proportion:	gap: 50.1%	gcc: 47.2%	overall: 48.65%
L/S proportion:	gap: 36.8%	gcc: 38.3%	overall: 37.55%
Cond Branch:	gap: 9.3%	gcc: 12.1%	overall: 10.7%
Jump:	gap: 4%	gcc: 1.9%	overall: 2.95%

overall CPI = (0.4865)(1.0 CPI) + (0.3755)(1.4 CPI) + (0.107)(0.6)(2.0 CPI) +
(0.107)(0.4)(1.5 CPI) + (0.0295)(1.2 CPI) = 1.24 CPI

5-A.3) Assuming the followings:

ALU: add, sub, mul, compare, load imm, cond move, shift, AND, OR, XOR, other logical

L/S: load, store

Cond Branch: cond branch

Jump: jump, call, return

FP multiply: mul FP

FP add: add FP

FP div: div FP

FP L/S: load FP, store FP

other FP: sub FP, move reg-reg FP, compare FP, cond mov FP, other FP

Proportions: ALU: 32.85% L/S: 12.2% Cond branch: 0.95%
Jump: 0% FP mul: 8.15% FP add: 8.6% FP div: 0.15%
FP L/S: 28.1% other FP: 9.1%

overall CPI = (0.3285)(1.0 CPI) + (0.122)(1.4 CPI) + (0.0815)(0.6)(2.0 CPI) +
(0.0095)(0.4)(1.5 CPI) + (0.0815)(6.0) + (0.086)(4.0 CPI) + (0.0015)(20.0 CPI) +
(0.281)(1.5) + (0.091)(2.0 CPI) = 1.9829 CPI

6-A.8) a) Yes, it is possible. Have all 0 address instructions begin with 111111, leaving 6 bits for the instruction opcode, all 1 address instructions begin with 11, leaving 5 bits for the address and 5 bits for the opcode, and have 2 address instructions start with 00, 01, or 10

b) No, it is not possible. This is because adding an extra instruction to the 1 address instructions would require all 0 address instructions to begin with 1111111, only leaving 5 bits for the opcode, so there can only be 32 instructions

c) There could still only be 31 one address instructions, because there are only 5 bits available for opcode after taking 5 for address and 2 for the two-address opcodes and the last option, 1111111 must be reserved to signify a zero-address opcode.

7-A.11)

32-bit size: (1 byte-char) + (1 byte-bool) + (2-bytes-waste) + (4 bytes-int) + (8 bytes-double) + (2 bytes-short) + (2 bytes-waste) + (4 bytes-float) + (8 bytes-double) + (4 bytes-char*) + (4 bytes-float*) + (4 bytes-int) = 44 bytes

rearranged: (1 byte-char) + (1-byte char) + (2 byte-short) + (4 byte-int) + (8 byte-double) + (8 byte-double) + (4 byte-float) + (4 byte-int) + (4 byte-char*) + (4 byte float*) = 40 bytes

64-bit: (1 byte-char) + (1-byte-bool) + (2 byte waste) + (4 byte int) + (8 byte double) + (2 byte short) + (2 byte empty) + (4 byte float) + (8 byte double) + (8 byte char*) + (8 byte float*) + (4 byte int) + (4 byte waste) = 56 bytes

by rearranging the 64-bit struct, it can be reduced to 48 bytes

using same instruction categories as A.1

A.20) ALU: 48% L/S: 36% Cond Branch: 12% Jump: 3%

a) ALU → 0 bits offset → 16 bit instruction

L/S → 30.4% 0bit, 41.2% 8 bit, 28.4% 16 bit

Cond Branch/Jump → 0.1% 0bit, 90.4% 8 bit, 9% 16 bit, 0.5% 24 bit

Average Length = (16bits)(0.48) + (16bits)(0.36)(0.304) + (16+8)(0.36)(0.412) + (16+16)(0.36)(0.284)
+ (16)(0.15)(0.001) + (16+8)(0.15)(0.904) + (16+16)(0.15)(0.09) + (16+24)(0.15)(0.005)
= 19.98 bits / 2.498 bytes 100 inst = 250 bytes

Assume offset of 0-8 bits = 1 instr, 9-16 bits = 2 inst, 17-24 bits = 3 instr

b) Number instructions: ALU – 100% 1 inst – Avg 3 bytes

L/S: 71.6% - 1 inst, 28.4% - 2 inst – Avg 3.852 bytes

Cond Branch/Jump: 90.5% - 1 inst, 9% - 2 inst, 0.5% - 3 inst – Avg 3.3 bytes

overall average: 3.32 byte

ratio: $\frac{\text{variable}}{\text{24-bit instr}} = \frac{3.32}{2.498} = 1.33$ 100 inst = 332 bytes

c) 100% of instructions are 5 bytes

100 inst = 500 bytes

ratio to 5 = $\frac{\text{40-bit instr}}{\text{24-bit instr}} = \frac{5 \text{ bytes}}{3.32 \text{ bytes}} = 1.51$