Julian Garcia

CSE331 Spring 2021
HW2

1. Compute the effective CPI for MIPS using Figure 1. Assume we have made the following measurements of average CPI for instruction types:

Instruction	Clock Cycles
All ALU instructions	1.0
Loads-stores	1.4
Conditional branches	
Taken	2.0
Not taken	1.5
Jumps	1.2

Assume that 60% of the conditional branches are taken and that all instructions in the "other" category of Figure 1 are ALU instructions. Average the instruction frequencies of gap and gcc to obtain the instruction mix.

Instruction	gap	gcc	gzip	mcf	perlbmk	Integer average
load	26.5%	25.1%	20.1%	30.3%	28.7%	26%
store	10.3%	13.2%	5.1%	4.3%	16.2%	10%
add	21.1%	19.0%	26.9%	10.1%	16.7%	19%
sub	1.7%	2.2%	5.1%	3.7%	2.5%	3%
mul	1.4%	0.1%				0%
compare	2.8%	6.1%	6.6%	6.3%	3.8%	5%
load imm	4.8%	2.5%	1.5%	0.1%	1.7%	2%
cond branch	9.3%	12.1%	11.0%	17.5%	10.9%	12%
cond move	0.4%	0.6%	1.1%	0.1%	1.9%	1%
jump	0.8%	0.7%	0.8%	0.7%	1.7%	1%
call	1.6%	0.6%	0.4%	3.2%	1.1%	1%
return	1.6%	0.6%	0.4%	3.2%	1.1%	1%
shift	3.8%	1.1%	2.1%	1.1%	0.5%	2%
AND	4.3%	4.6%	9.4%	0.2%	1.2%	4%
OR	7.9%	8.5%	4.8%	17.6%	8.7%	9%
XOR	1.8%	2.1%	4.4%	1.5%	2.8%	3%
other logical	0.1%	0.4%	0.1%	0.1%	0.3%	0%
load FP						0%
store FP						0%
add FP						0%
sub FP						0%
mul FP						0%
div FP						0%
mov reg-reg FP						0%
compare FP						0%
cond mov FP						0%
other FP						0%

Figure 1. **MIPS dynamic instruction mix for five SPECint2000 programs.** Note that integer register-register move instructions are included in the OR instruction. Blank entries have the value 0.0%.

Averaging gap & gcc:
load = (26.5 + 25.1)/2 = 25.8%
store = (10.3 + 13.2)/2 = 11.75%
add = (21.1 + 19.0)/2 = 20.05%
sub = (1.7 + 2.2)/2 = 1.95%
mul = (1.4 + 0.1)/2 = 0.75%
compare = (2.8 + 6.1)/2 = 4.45%
load imm = (4.8 + 2.5)/2 = 3.65%
cond branch = (9.3 + 12.1)/2 = 10.7%
cond move = (0.4 + 0.6)/2 = 0.5%
jump = (0.8 + 0.7)/2 = 0.75%
call = (1.6 + 0.6)/2 = 1.1%
return = (1.6 + 0.6)/2 = 1.1%
shift = (3.8 + 1.1)/2 = 2.45%
AND = (4.3 + 4.6)/2 = 4.45 %
OR = (7.9 + 8.5)/2 = 8.2 %
XOR = (1.8 + 2.1)/2 = 1.95 %
Other = (0.1 + 0.4)/2 = 0.25 %

Calculating CPI using instruction Mix:
ALL ALU INSTRUCTIONS = 1.0 * (20.5 + 1.95 + 0.75 + 2.45 + 4.45 + 4.45 + 1.95 + 0.25)/100 = 0.3675

LOADS-STORES = 1.4 * (25.8 + 11.75 + 3.65) / 100 = 0.5768
CONDITIONAL BRANCHES:
TAKEN = (2.0 * 0.6) * (10.7 + 0.5) / 100 = 0.1344
NOT TAKEN =(1.5 * 0.4) * (10.7 + 0.5) / 100 = 0.0672
JUMPS = 1.2 * (0.75 + 1.1 + 1.1) / 100 = 0.0354

OVERALL CPI = 0.3675 + 0.5768 + 0.1344 + 0.0672 + 0.0354 = 1.1813

2. For the following we consider instruction encoding for instruction set architectures.
a. Consider the case of a processor with an instruction length of 12 bits and with 32 general-purpose registers so the size of the address fields is 5 bits. Is it possible to have instruction encodings for the following?
■ 3 two-address instructions
■ 30 one-address instructions
■ 45 zero-address instructions
3 two-address instructions: 3 * 2^5 * 2^5 possible combinations = 3072
30 one-address instructions: 30 * 2^5 = 960
45 zero-address instructions: 45
total = 3072 + 960 + 45 = 4077
2^12 == 4096, and 4077 < 4096, so it should be possible.

b. Assuming the same instruction length and address field sizes as above, determine if it is possible to have
■ 3 two-address instructions
■ 31 one-address instructions
■ 35 zero-address instructions

3 two-address instructions: 3 * 2^5 * 2^5 possible combinations = 3072
31 one-address instructions: 30 * 2^5 = 992
35 zero-address instructions: 35
total = 3072 + 992 + 35 = 4099
2^12 == 4096, and 4099 > 4096, so it should NOT be possible

Explain your answer.
Part b is not possible to have instruction encodings for because it does not fit into the instruction length, as it has a size of 4096 which is less than the size needed of 4099.

c. Assume the same instruction length and address field sizes as above. Further assume there are already 3 two-address and 24 zero-address instructions. What is the maximum number of one-address instructions that can be encoded for this processor?
3 two-address instructions: 3 * 2^5 * 2^5 possible combinations = 3072
24 zero-address instructions: 24
Total = 3096
Available Space = 2^12 - 3096 = 4096 - 3096 = 1000
1 one-address instruction = 1 * 2^5 = 32
Max number of one address instructions = 1000/32 = 31.25 (ROUNDED TO) 31

The Maximum number of one-address instructions that can be encoded for this processor is 31.

3. For the following assume that values A, B, C, D, E, and F reside in memory. Also assume that instruction operation codes are represented in 8 bits, memory addresses are 64 bits, and register addresses are 6 bits.
a. For each instruction set architecture shown in Figure 2, how many addresses, or names, appear in each instruction for the code to compute C = A + B, and what is the total code size?

Architecture type	Stack	Accumulator	Mem-mem Register	Load-Store Register
# Addresses	3	3	7	9
Code size	224 bits	216 bits	240 bits	260 bits

b. Some of the instruction set architectures in Figure 2 destroy operands in the course of computation. This loss of data values from processor internal storage has performance consequences. For each architecture in Figure 2, write the code sequence to compute:
C = A + B
D = A – E
F = C + D
In your code, mark each operand that is destroyed during execution and mark each "overhead" instruction that is included just to overcome this loss of data from processor internal storage. What is the total code size, the number of bytes of instructions and data moved to or from memory, the number of overhead instructions, and the number of overhead data bytes for each of your code sequences?

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C

Figure 2. The code sequence for C = A + B for four classes of instruction sets. Note that the Add instruction has implicit operands for stack and accumulator architectures and explicit operands for register architectures. It is assumed that A, B, and C all belong in memory and that the values of A and B cannot be destroyed.

STACK:	MARKS	BITS
Push A		8+64
Push B		8+64
Add (A,B)	DESTROYED OPERAND	8
Pop C (C)	DESTROYED OPERAND	8+64
Push A	OVERHEAD INSTRUCTION	8+64
Push E		8+64
Sub (A,E)	DESTROYED OPERAND	8
Pop D (D)	DESTROYED OPERAND	8+64
Push C	OVERHEAD INSTRUCTION	8+64
Push D	OVERHEAD INSTRUCTION	8+64
Add (C, D)	DESTROYED OPERAND	8
Pop F (F)	DESTROYED OPERAND	8+64

CODE SIZE: 672 bits
DATA MOVED = 36Bytes
Overhead data bytes = 3*4 = 12 bytes

Julian Garcia

Accumulator:	MARKS	BITS
Load A		8+64
Add B (A)	DESTROYED OPERAND	8+64
Store C		8+64
Load A (C)	DEST. OP./OVERHEAD INST.	8+64
Sube E (A)	DESTROYED OPERAND	8+64
Store D		8+64
Add C	OVERHEAD INSTRUCTION	8+64
Store F		8+64

CODE SIZE: 576bits
DATA MOVED = 28Bytes
Overhead data bytes = 2*4 = 8 bytes

Register (reg-mem):	MARKS	BITS
Load R1, A		8+6+64
Add R2, R1, B		8+6+6+64
Store R2, C		8+6+64
Sub R3, R1, E		8+6+6+64
Store R3, D		8+6+64
Add R4, R3, C	OVERHEAD INSTRUCTION	8+6+6+64
Store R4, F		8+6+64

CODE SIZE: 564bits
DATA MOVED = 28Bytes
Overhead data bytes = 1*4 = 4 bytes

Register (load-store):	MARKS	BITS
Load R1, A		8+6+64
Load R2, B		8+6+64
Add R3, R1, R2		8+6+6+6
Store R3, C		8+6+64
Load R4, E		8+6+64
Sub R5, R1, R4		8+6+6+6
Store R5, D		8+6+64
Add R6, R3, R5		8+6+6+6
Store R6, F		8+6+64

CODE SIZE: 546bits
DATA MOVED = 24Bytes
Overhead data bytes = 0 Bytes