COE 501: Computer Architecture

Problem Set 2: Instruction Set Principles

Solution

1) (4 pts) The MIPS dynamic instruction mix for the gcc program is given below:

load	store	add	sub	compare	cond branch	jump	call	return	shift	and	or	xor	other ALU
25.1%	13.2%	19.0%	2.2%	6.1%	12.1%	0.7%	0.9%	0.9%	1.1%	4.6%	8.5%	2.1%	3.5%

Given the following measurements of average CPI for instruction types:

Instruction Type	All ALU Instructions	Load & Store	Taken Branch	Not Taken Banch	Jump, call, and return
Average CPI	1.0	1.5	1.8	1.2	1.3

Assume that 70% of the conditional branches are taken, compute the effective CPI for the above gcc program.

Total ALU instructions = 19.0% + 2.2% + 6.1% + 1.1% + 4.6% + 8.5% +2.1% + 3.5% = 47.1%
Load and Store = 25.1% + 13.2% = 38.3%
Taken Branch = 12.1% × 0.7 = 8.47%
Not Taken Branch = 12.1% × 0.3 = 3.63%
Jump, Call, and Return = 0.7% + 0.9% + 0.9% = 2.5%

Effective CPI = 0.471 × 1.0 + 0.383 × 1.5 + 0.0847 × 1.8 + 0.0363 × 1.2 + 0.025 × 1.3 = 1.274

2) (12 pts) Consider the following for-loop:

```
for (i = 0; i < 100; i++) { A[i] = B[i] + C; }
```

Where **A** and **B** are arrays of 64-bit integers, and **C** and **i** are 64-bit integers. Array **A** starts at address 1000 in memory, array **B** starts at address 3000, and variable **C** is at address 5000. The value of variable **C** should be loaded once into a register (**R8**) before entering the loop, and used in all loop iterations. Allocate a register for variable **i** (**R9**), and do not load or store its value in memory.

a) (6 pts) Write the code for MIPS64.

```
    LD      R8,  5000(R0)       ; Load R8 = C
    DADD    R9,  R0, R0         ; R9 = i = 0
    ORI     R10, R0, 1000       ; R10 = address of array A
    ORI     R11, R0, 3000       ; R11 = address of array B
    ORI     R15, R0, 100        ; R15 = 100
loop:
    LD      R12, (R11)          ; Load R12 = B[i]
    DADD    R13, R12, R8        ; R13 = B[i] + C
    SD      R13, (R10)          ; Store A[i] = R13
    DADD    R10, R10, 8         ; R10 = address of next A[i]
    DADD    R11, R11, 8         ; R11 = address of next B[i]
    DADD    R9,  R9,  1         ; i++
    BNE     R9,  R15, loop      ; branch if (i != 100)
```

b) (2 pts) How many instructions are executed dynamically?

According to the code provided in part a:
Total instructions executed = 5 (outside loop) + 7 × 100 (inner loop) = 705

c) (2 pts) How many load/store instructions are executed dynamically?

Total load/store executed = 1 (outside loop) + 2 × 100 (inner loop) = 201

d) (2 pts) What is the code size in bytes?

Code size in bytes = 12 instructions × 4 bytes = 48 bytes

3) (12 pts) Assume that variables **A** thru **F** reside in memory and their memory addresses reside in registers **R1** to **R6**, respectively. Instruction opcodes are 8 bits and register addresses are 4 bits. Instructions can vary in size according to the opcode.

a) (8 pts) Consider four different classes of instruction set architectures: Accumulator, Stack, Register-Memory (2-address instructions), and Register-Register (3-address instructions). For each instruction set class, write the code sequence to compute:

```
C = A + B
D = A – E
F = C + D
```

b) (4 pts) What is the code size (bits) for each instruction set class?

Solution for parts a and b:
The code is shown under the first column and the code size (in bits) is shown under the second column for each instruction set class.

Accumulator		Stack		Register-Memory		Register-Register	
Code	**Size**	**Code**	**Size**	**Code**	**Size**	**Code**	**Size**
Load (R1)	12	Push (R1)	12	Load R7,(R1)	16	Load R7,(R1)	16
Add (R2)	12	Push (R2)	12	Add R7,(R2)	16	Load R8,(R2)	16
Store (R3)	12	Add	8	Store R7,(R3)	16	Add R9,R7,R8	20
Load (R1)	12	Pop (R3)	12	Load R8,(R1)	16	Store R9,(R3)	16
Sub (R5)	12	Push (R5)	12	Sub R8,(R5)	16	Load R10,(R5)	16
Store (R4)	12	Push (R1)	12	Store R8,(R4)	16	Sub R11,R7,R10	20
Add (R3)	12	Sub	8	Add R7,R8	16	Store R11,(R4)	16
Store (R6)	12	Pop (R4)	12	Store R7,(R6)	16	Add R12,R9,R11	20
		Push (R3)	12			Store R12,(R6)	16
		Push (R4)	12				
		Add	8				
		Pop (R6)	12				
Total	**96 bits**		**132 bits**		**128 bits**		**156 bits**

4) (8 pts) Consider a C structure (Object) that includes the following members. Assume a 64-bit architecture with 64-bit memory addresses:

```
struct Object {
  bool b;
  double d;
  short s1;
  char * cptr;   // pointer to char
  short s2;
  short s3;
  float * fptr;  // pointer to float
  float f;
  int i;
};
```

a) (2 pts) If the compiler does not align data in memory, what is the minimum size of the struct Object in bytes?

Minimum Size (no alignment) = 1 + 8 + 2 + 8 + 2 + 2 + 8 + 4 + 4 = 39 bytes

b) (3 pts) If the compiler aligns data in memory according to their size, but does not reorder the members of the structure, what will be the size of the struct Object? How many bytes are wasted internally inside the structure due to memory alignment?

Size with alignment = 1 + skip 7 + 8 + 2 + skip 6 + 8 + 2 + 2 + skip 4 + 8 + 4 + 4 = 56 bytes
7 + 6 + 4 = 17 bytes are skipped (wasted) internally due to memory alignment

c) (3 pts) If the compiler aligns data in memory and can also reorder the members of the structure, how the members of the struct Object should be reordered? What will be the size of the struct Object?

Members can be sorted according to their size

```
struct Object {
  bool b;
  short s1;
  short s2;
  short s3;
  float f;
  int i;
  double d;
  char * cptr;  // pointer to char
  float * fptr; // pointer to float
};
```

Size after reordering = 1 + skip 1 + 2 + 2 + 2 + 4 + 4 + 8 + 8 + 8 = 40 bytes

5) (14 pts) The MIPS architecture divides registers into two groups: Caller-Saved and Callee-Saved. Those registers that are **caller-saved** should be preserved by the caller (if needed) before making a procedure call. Those registers that are **callee-saved** should be preserved inside the called procedure (if needed) before modifying them. Registers are saved by storing their values on the runtime stack and restored by loading their values.

Register Number	Usage	Preserved by
R0	Always zero	Not applicable
R1	Reserved for assembler use	Not applicable
R2 – R3	For returning function results	Caller
R4 – R7	For passing arguments	Caller
R8 – R15	For temporary results	Caller
R16 – R23	For saved variables	Inside procedure (Callee)
R24 – R25	More temporary results	Caller
R26 – R27	OS kernel	Not applicable
R28	Global data pointer	Inside procedure (Callee)
R29	Stack pointer	Inside procedure (Callee)
R30	Frame pointer	Inside procedure (Callee)
R31	Return address	Inside procedure (Callee)

Suppose we want to translate function **f** using the above MIPS register conventions. Function **f** receives four arguments in registers **R4** thru **R7** and returns its result in register **R2**. We know that function **f** calls **g**, but we do not have the code for **g** and we don't know which registers are used in **g**. We only know that **g** receives two integer arguments and returns one integer result according to the MIPS register conventions shown above.

a) (5 pts) Translate function **f** into MIPS64 code:

```
int f(int a, int b, int c, int d) {
  return g(g(a,b), c+d);
}
```

```
f:  DADDI  R29, R29, -16  ; Decrement stack pointer by 16 bytes
    SD     R31, (R29)     ; Save return address on the stack
    SD     R16, 8(R29)    ; Save R16 on the stack
    DADD   R16, R6, R7    ; R16 = c+d
    JAL    g              ; Call function g; R2 = result
    ORI    R4, R2, 0      ; R4 = R2 = result of first call
    ORI    R5, R16, 0     ; R5 = R16 = c+d
    JAL    g              ; Second call to g; R2 = result
    LD     R31, (R29)     ; Load return address on the stack
    LD     R16, 8(R29)    ; Restore R16 from the stack
    DADDI  R29, R29, 16   ; Free space on the stack
    JR     R31            ; Return to caller
```

b) (5 pts) Translate function **h** into MIPS64 code:

```
int h(int a, int b, int c, int d) {
  if (a+b<c+d) return g(a+b, c+d);
  else return g(c+d, a+b);
}
```

```
h:  DADDI  R29, R29, -8   ; Decrement stack pointer by 8 bytes
    SD     R31, (R29)     ; Save return address on the stack
    DADDI  R4, R4, R5     ; R4 = a+b
    DADDI  R5, R6, R7     ; R5 = c+d
    SLT    R8, R4, R5     ; R8 = (a+b<c+d)
    BEQ    R8, R0, else   ; Branch if false to else part
    JAL    g              ; Call g(a+b, c+d); R2 = result
    J      return
else:
    ORI    R9, R4, 0      ; R9 = R4 = a+b
    ORI    R4, R5, 0      ; R4 = R5 = c+d
    ORI    R5, R9, 0      ; R5 = R9 = a+b
    JAL    g              ; Call g(c+d, a+b); R2 = result
return:
    LD     R31, (R29)     ; Load return address on the stack
    DADDI  R29, R29, 8    ; Free space on the stack
    JR     R31            ; Return to caller
```

c) (4 pts) Can we use tail-call optimization that uses the **J** (jump) instruction rather than **JAL** (jump-and-link) to call the function **g** in part a and b? If no, explain why not. If yes, show how the code should be optimized in part a and b?

For part a, we can use tail-call optimization for the second call to g. However, we must save the return for the first call to g. The code for part a is as follows:

```
f:  DADDI  R29, R29, -16  ; Decrement stack pointer by 16 bytes
    SD     R31, (R29)     ; Save return address on the stack
    SD     R16, 8(R29)    ; Save R16 on the stack
    DADD   R16, R6, R7    ; R16 = c+d
    JAL    g              ; Call function g; R2 = result
    ORI    R4, R2, 0      ; R4 = R2 = result of first call
    ORI    R5, R16, 0     ; R5 = R16 = c+d
    LD     R31, (R29)     ; Load return address on the stack
    LD     R16, 8(R29)    ; Restore R16 from the stack
    DADDI  R29, R29, 16   ; Free space on the stack
    J      g              ; Jump to g (tail-call); R2 = result
```

For part b, there is only one call to g (either IF part or ELSE part). We can use tail-call optimization as follows: (No need to save return address on the stack)

```
h:  DADDI  R4, R4, R5     ; R4 = a+b
    DADDI  R5, R6, R7     ; R5 = c+d
    SLT    R8, R4, R5     ; R8 = (a+b<c+d)
    BEQ    R8, R0, else   ; Branch if false to else part
    J      g              ; Tail-Call g(a+b, c+d); R2 = result
else:
    ORI    R9, R4, 0      ; R9 = R4 = a+b
    ORI    R4, R5, 0      ; R4 = R5 = c+d
    ORI    R5, R9, 0      ; R5 = R9 = a+b
    J      g              ; Tail-Call g(c+d, a+b); R2 = result
```