ECSE425 Computer Architecture and Organization Assignment 2 Solutions

Linda Wang

Fall 2008

Question 1: Tradeoffs

Before the enhancement, the CPI is

$$\text{CPI}_{\text{old}} = 0.05 \times 4.0 + 0.30 \times 1.5 + 0.45 \times 1.2 + 0.20 \times 2.0 = 1.590.$$

After the enhancement, half of the fixed-point multiplication instructions are eliminated, and each instruction eliminated is replaced by two ALU instructions. Therefore the relationship between the old and new instruction counts is

$$\text{IC}_{\text{new}} = \left(1 - \frac{0.05}{2} + \frac{0.05}{2} \times 2\right) \text{IC}_{\text{old}} = 1.025\text{IC}_{\text{old}}.$$

And the new instruction frequencies are

Type	Frequency	CPI
fixed-point mult	0.025/1.025	4.0
load/store	0.30/1.025	1.5
ALU	(0.45+0.05)/1.025	1.2
others	0.20/1.025	2.0

So the new CPI is

$$\text{CPI}_{\text{new}} = \frac{1}{1.025}(0.025 \times 4.0 + 0.30 \times 1.5 + 0.50 \times 1.2 + 0.20 \times 2.0) = 1.512.$$

Finally, the clock cycle time does not change, so $\text{CC}_{\text{new}} = \text{CC}_{\text{old}}$. The overall speedup is

$$\text{speedup} = \frac{\text{IC}_{\text{old}} \times 1.590 \times \text{CC}_{\text{old}}}{1.025\text{IC}_{\text{new}} \times 1.512 \times \text{CC}_{\text{new}}} = 1.026.$$

Question 2: CPI

The following table shows the instruction mix taken from the text. Note that "ALU" includes add, sub, mul, compare, load imm, shift, and, or, xor, and other logical instructions.

Type	Frequency
ALU	47%
load/store	36%
branches	12%
jumps	1%
others	4%

The effective CPI for the MIPS machine is

$$\text{CPI} = 0.47 \times 1.4 + 0.36 \times 1.6 + 0.12 \times 2.8 + 0.01 \times 1.2 + 0.04 \times 2.4 = 1.678.$$

Question 3: MIPS64 machine code

For both program, it is assumed that we are *not* using delayed branches. Also, we ignore call/return and stack pointer adjustment codes.

```
(a)
;---------------------------------------------------------------
; Register assignments:
;  R1 = i
;  R2 = address of A
;  R3 = c
; Assume that A is an array of integers
;---------------------------------------------------------------

        DADD    R1,R0,R0        ; i = 0
loop:   ANDI    R2,R1,#1        ; R2 = i AND 1
        BNEZ    R2,skip         ; if i is not even, go to skip
        DSLL    R4,R1,#2        ; R4 = i*4 (integers are 4 bytes)
        DADD    R5,R2,R4        ; R5 = address of a[i]
        LW      R6,0(R5)        ; R6 = a[i]
        DSUB    R6,R6,R3        ; R6 = a[i] - c
        SW      R6,0(R5)        ; store new value of a[i]
        DADDI   R3,R3,#1        ; c++
skip:   DADDI   R1,R1,#1        ; i++
        SLTI    R7,R1,#100      ; R7 = (i<100)
        BNEZ    R7,loop         ; if i<100 go to loop

(b)
;---------------------------------------------------------------------
; Register assignments:
;  R1 = dst
```

```
;  R2 = src
;  R3 = size (assumed to be equal to sizeof(dst), which is NOT the
;       same as strlen(dst)!)
;  R4 = return value (length of final string)
;  R5 = array index i (used to index dst)
;  R6 = array index j (used to index src)
;
; The implementation must satisfy the following:
; (1) The max # of chars copied from src to dst is size-strlen(dst)-1
; (2) The function returns the # chars of the final string
; (3) The function always NUL-terminates the final string, unless (4)
;     occurs
; (4) If 'size' chars have been traversed in dst without finding a NUL,
;     then the function returns 'size' and no NUL-termination is done
;-------------------------------------------------------------------

;---- initialize
        DADD    R5,R0,R0        ; i = 0
        DADD    R6,R0,R0        ; j = 0
        DADDI   R9,R3,#-1       ; R9 = last index of dst

;---- traverse dst and find NUL
loop1:  BEQ     R5,R3,exit2     ; if size chars are traversed w/o
                                ; finding NUL, go to exit2
        DADD    R7,R5,R1        ; R7 = address of dst[i]
        LB      R8,0(R7)        ; R8 = dst[i]
        BEQZ    R8,loop2        ; if dst[i]=NUL, start concatenation
        DADDI   R5,R5,#1        ; i++
        J       loop1

;---- append src to dst
loop2:  BEQ     R5,R9,exit      ; if end of dst reached, go to exit
        DADD    R10,R6,R2       ; R10 = address of src[j]
        LB      R11,0(R10)      ; R11 = src[j]
        DADD    R7,R5,R1        ; R7 = address of dst[i]
        SB      R11,0(R7)       ; dst[i] = src[j]
        BEQ     R11,R0,exit     ; if src[j]=NUL, go to exit2
        DADDI   R5,R5,#1        ; i++
        DADDI   R6,R6,#1        ; j++
        J       loop2

;---- append NUL, put return value in R6, and exit
exit:   DADD    R7,R5,R1        ; address of dst[i]
        SB      R0,0(R7)        ; dst[i] = NUL
exit2:  DADD    R4,R5,R0        ; return value = i
```

Question 4: ISA analysis

- The processor has 32 general purpose registers, 32 floating-point registers, many special-purpose registers, and a few miscellaneous registers (pages 1-35 to 1-36).
- Its instruction types are: integer ALU/logical instructions, floating-point ALU/logical instructions, load and store instructions, flow control instructions, processor control instructions, and memory control instructions (pages 1-45 to 1-46).
- Its integer instruction types are: arithmetic instructions, compare instructions, logical instructions, rotate and shift instructions, load and store instructions (page 1-45).
- It has fixed-length (32-bit) instruction encoding (page 1-45).
- The effective address is 32 bits (page 2-63).
- Memory operands can be bytes, half words (16 bits), words (32 bits), and double words (64 bits). In addition, for the load/store multiple and load/store string instructions, the memory operand can be a sequence of bytes or words (page 2-63).
- Floating-point load/store instructions must operate on aligned memory operands (page 2-58).
- The two addressing modes for load/store are: register indirect with immediate index (called "displacement" in Figure 2.6 of text), and register indirect with index (called "indexed" in Figure 2.6 of text) (page 2-64).
- The register has 32 bits. It is called the Condition Register (CR) (page 1-39).

Question 5: IC's and mixes

The assembly output of both the unoptimized and the optimized code are shown below with comments added (note that different versions of gcc may produce somewhat different assembly outputs). In order to count the number of instructions, it is important to understand how the assembly output corresponds to the original C code.

Assembly output of unoptimized code

```
        .file        "M.c"
gcc2_compiled.:
.section        ".rodata"
        .align 8
.LLC0:
        .uaword        0x0 ! ~0.0000000000000000000e0
        .uaword 0x0
.section        ".text"
        .align 4
        .global mmult
```

```
        .type   mmult,#function
        .proc   020
mmult:
        !#PROLOGUE# 0
        save    %sp, -136, %sp
        !#PROLOGUE# 1
        nop

        st      %g0, [%fp-20]   ; i=0

        ;; ---- i loop
.LL3:
        ld      [%fp-20], %o0   ; load i
        cmp     %o0, 19         ; compare i with 19
        ble     .LL6            ; if i<=19 go to j loop
        nop                     ; delay slot
        b       .LL4            ; otherwise return
        nop                     ; delay slot
.LL6:
        st      %g0, [%fp-24]   ; j=0

        ;; ---- j loop
.LL7:
        ld      [%fp-24], %o0   ; load j
        cmp     %o0, 19         ; compare j with 19
        ble     .LL10           ; if i<=19 go to k loop
        nop                     ; delay slot
        b       .LL5            ; otherwise go to end of i loop
        nop                     ; delay slot
.LL10:
        st      %g0, [%fp-28]   ; k=0
        sethi   %hi(.LLC0), %o1
        or      %o1, %lo(.LLC0), %o0
        ldd     [%o0], %o2
        std     %o2, [%fp-40]   ; sum = 0.0

        ;; ---- k loop
.LL11:
        ld      [%fp-28], %o0   ; load k
        cmp     %o0, 19         ; compare k with 19
        ble     .LL14           ; if k<=19 go to body of loop
        nop                     ; delay slot
        b       .LL9            ; otherwise go to end of j loop
        nop                     ; delay slot
.LL14:
        ;; ---- get address of A[i][k]
        sethi   %hi(A), %o1
```

```
or      %o1, %lo(A), %o0
        ld      [%fp-28], %o1
        mov     %o1, %o2
        sll     %o2, 3, %o1
        ld      [%fp-20], %o2
        mov     %o2, %o4
        sll     %o4, 2, %o3
        add     %o3, %o2, %o3
        sll     %o3, 5, %o2
        add     %o1, %o2, %o1

        ;; ---- get address of B[k][j]
        sethi   %hi(B), %o3
        or      %o3, %lo(B), %o2
        ld      [%fp-24], %o3
        mov     %o3, %o4
        sll     %o4, 3, %o3
        ld      [%fp-28], %o4
        mov     %o4, %g2
        sll     %g2, 2, %o5
        add     %o5, %o4, %o5
        sll     %o5, 5, %o4
        add     %o3, %o4, %o3

        ;; ---- load A[i][k]
ldd     [%o0+%o1], %f4
        fmovs   %f4, %f2
        fmovs   %f5, %f3

        ;; ---- load B[k][j]
ldd     [%o2+%o3], %f6
        fmovs   %f6, %f4
        fmovs   %f7, %f5

        ;; ---- sum += A[i][k] * B[k][j]
fmuld   %f2, %f4, %f2   ; multiply
        ldd     [%fp-40], %f4   ; load sum
        faddd   %f4, %f2, %f2   ; add
        std     %f2, [%fp-40]   ; store sum

        ;; ---- get address of C[i][j]
        sethi   %hi(C), %o1
        or      %o1, %lo(C), %o0
        ld      [%fp-24], %o1
        mov     %o1, %o2
        sll     %o2, 3, %o1
        ld      [%fp-20], %o2
```

```
        mov     %o2, %o4
        sll     %o4, 2, %o3
        add     %o3, %o2, %o3
        sll     %o3, 5, %o2
        add     %o1, %o2, %o1

        ;; ---- C[i][j] = sum
ldd     [%fp-40], %o2   ; load sum
        mov     %o2, %o4
        mov     %o3, %o5
        std     %o4, [%o0+%o1]  ; store to C[i][j]

        ;; ---- end of k loop
.LL13:
ld      [%fp-28], %o0   ; load k
        add     %o0, 1, %o1     ; k++
        st      %o1, [%fp-28]   ; store k
        b       .LL11           ; jump to start of k loop
        nop                     ; delay slot
.LL12:
        ;; ---- end of j loop
.LL9:
        ld      [%fp-24], %o0   ; load j
        add     %o0, 1, %o1     ; j++
        st      %o1, [%fp-24]   ; store j
        b       .LL7            ; jump to start of j loop
        nop                     ; delay slot
.LL8:
        ;; ---- end of i loop
.LL5:
        ld      [%fp-20], %o0   ; load i
        add     %o0, 1, %o1     ; i++
        st      %o1, [%fp-20]   ; store i
        b       .LL3            ; jump to start of i loop
        nop                     ; delay slot
.LL4:
        ;; ---- return from function
.LL2:
        ret
        restore
.LLfe1:
        .size   mmult,.LLfe1-mmult
        .common A,3200,8
        .common B,3200,8
        .common C,3200,8
        .ident  "GCC: (GNU) 2.95.3 20010315 (release)"
```

Assembly output of optimized code

```
        .file "M.c"
gcc2_compiled.:
        .common A,3200,8
        .common B,3200,8
        .common C,3200,8
.section        ".rodata"
        .align 8
.LLC1:
        .uaword 0x0 ! ~0.0000000000000000000e0
        .uaword 0x0
.section        ".text"
        .align 4
        .global mmult
        .type   mmult,#function
        .proc   020
mmult:
        !#PROLOGUE# 0
        !#PROLOGUE# 1

        ;; ---- initialize
        sethi   %hi(A), %g2
        or      %g2, %lo(A), %g4 ; get base addr of A
        sethi   %hi(B), %g3
        sethi   %hi(C), %g2
        or      %g3, %lo(B), %g3 ; get base addr of B
        or      %g2, %lo(C), %g1 ; get base addr of C
        mov     0, %o1           ; i = 0
        sll     %o1, 2, %g2

        ;; ---- i loop
.LL19:
        add     %g2, %o1, %g2
        sll     %g2, 5, %o2
        mov     0, %o0                  ; j = 0
        add     %o1, 1, %o5             ; i++
        sll     %o0, 3, %g2

        ;; ---- j loop
.LL18:
        add     %o0, 1, %o4             ; j++
        sethi   %hi(.LLC1), %o0
        add     %g2, %o2, %o3
        ldd     [%o0+%lo(.LLC1)], %f6   ; sum = 0.0
        add     %g2, %g3, %g2           ; get addr of B[k][j]
        add     %o2, %g4, %o0           ; get addr of A[i][k]
        mov     19, %o1                 ; k = 19
```

```
        ;; ---- k loop
.LL14:
        ldd     [%g2], %f2      ; load B[k][j]
        addcc   %o1, -1, %o1    ; k--
        ldd     [%o0], %f4      ; load A[i][k]
        add     %g2, 160, %g2   ; get addr of B[k+1][j]
        fmuld   %f4, %f2, %f4   ; A[i][k]*B[k][j]
        add     %o0, 8, %o0     ; get addr of A[i][k+1]
        faddd   %f6, %f4, %f6   ; update sum
        fmovs   %f6, %f2
        fmovs   %f7, %f3

        ;; ---- end of k loop
        bpos    .LL14           ; if k>0 go to start of k loop
        std     %f2, [%g1+%o3]  ; delay slot: C[i][j]=sum

        ;; ---- end of j loop
mov     %o4, %o0
        cmp     %o0, 19         ; compare j with 19
        ble,a   .LL18           ; if j<=19 go to start of j loop
        sll     %o0, 3, %g2     ; delay slot

        ;; ---- end of i loop
mov     %o5, %o1
        cmp     %o1, 19         ; compare i with 19
        ble     .LL19           ; if i<=19 go to start of i loop
        sll     %o1, 2, %g2     ; delay slot

        ;; ---- return
        retl
        nop
.LLfe1:
        .size    mmult,.LLfe1-mmult
        .ident  "GCC: (GNU) 2.95.3 20010315 (release)"
```

Instruction count of unoptimized and optimized code

The following table shows the instruction count per loop iteration for both the unoptimized and the optimized code. The i loop is iterated 50 times, the j loop is iterated $20^2 = 400$ times, and the k loop is iterated $20^3 = 8000$ times. Empty entries in the table indicate zeros.

# of instructions per i, j, and k loop iteration

Instr	Unoptimized			Optimized		
Type	i loop	j loop	k loop	i loop	j loop	k loop
ld	2	2	8			
st	2	2	1			
ldd		1	4		1	2
std		1	2			1
cmp	1	1	1	1	1	
add	1	1	7	2	4	2
addcc						1
or		1	3			
sll			9	3	1	
sethi		1	3		1	
fmuld			1			1
faddd			1			1
b	2	2	1			
ble	1	1	1	1	1	
bpos						1
mov			8	2	2	
fmovs			4			2
nop	3	3	2			
Total	12	16	56	9	11	11

So the total instruction counts are approximately:

$$\text{IC}_{\text{unopt}} = 20 \times 12 + 400 \times 16 + 8000 \times 56 = 454,640,$$
$$\text{IC}_{\text{opt}} = 20 \times 9 + 400 \times 11 + 8000 \times 11 = 92,580.$$

We can see that unoptimized code produces $\frac{454640}{92580} = 5$ times more instructions than optimized code. Note that this way of counting instructions is just a rough approximation (for example, some instructions may not be executed exactly 20 times; other instructions that execute only once are not counted). We just need a ballpark figure for comparison.

Instruction mix of unoptimized and optimized code

The instructions can be divided into the following types: integer load/store (ld and st), floating point load/store (ldd and std), integer ALU/logical (cmp, add, addcc, or, sll, sethi), floating point ALU (fmuld and faddd), branch (ble and bpos), jump (b), mov (mov and fmovs), and nop. The following table shows the instruction count and the frequency of the instruction types.

Instrucion mix of unoptimized and optimized code

Instr Type	Unoptimized		Optimized	
	#Instructions	Frequency	#Instructions	Frequency
int load/store	73680	16.21%	0	0%
FP load/store	48800	10.73%	24400	26.36%
int ALU/logical	185640	40.83%	26920	29.08%
FP ALU	16000	3.52 %	16000	17.28%
branch	8840	1.94 %	8420	9.09%
jump	8420	1.85 %	0	0%
mov	96000	21.12%	16200	17.50%
nop	17260	3.80 %	0	0%

Optimization has resulted in much more efficient access of the array elements. Clever schemes for calculating array element addresses have removed all integer loads and stores, and greatly reduced integer ALU/logical and mov operations. Also, by moving loop termination checks to be later in the code, all unconditional jumps are eliminated. Finally, optimization has filled in the branch delay slots with useful instructions instead of nops.