Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Fall Term 2012

SYSTEMS PROGRAMMING AND COMPUTER ARCHITECTURE
Assignment 5: Sequential processors

Assigned on: **1st Nov 2012**
Due by: **8th Nov 2012**

Part 1: Paper Exercises

A simple C program

Consider the following C program that is *manually* compiled into Y86 assembly.

```
typedef enum {ADD=0, SUB=1, AND=2, XOR=3} op_t;
void calc(op_t op, int a, int b, int *result) {
   if (op>3) return;
   switch(op) {
   case ADD: *result = a+b; break;
   case SUB: *result = a-b; break;
   case AND: *result = a&b; break;
   case XOR: *result = a^b; break;
   }
}
op_t op = SUB;
int  a = 37, b = 4, c;
main() {
   calc(op,a,b,&c);
}
```

Output of Y86 Assembler

Shown below is the "object code" of the Y86 program, generated by the Y86 assembler.

```
                     | # Execution starts at address 0
  0x000:             |          .pos 0
  0x000: 308400100000 | init:    irmovl   stack, %esp
  0x006: 308500100000 |          irmovl   stack, %ebp
  0x00c: 7034000000   |          jmp      main
                     |
                     |
                     | # global variables
  0x014:             |          .align   4
  0x014: 01000000    | op:      .long    0x1            # operation
  0x018: 25000000    | varA:    .long    0x25           # variable a
  0x01c: 04000000    | varB:    .long    0x4            # variable b
  0x020: 00000000    | varC:    .long    0x0            # variable c
                     |
  0x024: 94000000    | table:   .long    addop
  0x028: 9b000000    |          .long    subop
  0x02c: a2000000    |          .long    andop
  0x030: a9000000    |          .long    xorop
```

```
                        |
  0x034: 308020000000 | main:    irmovl   varC,%eax
  0x03a: a008         |          pushl    %eax
  0x03c: 50081c000000 |          mrmovl   varB,%eax
  0x042: a008         |          pushl    %eax
  0x044: 500818000000 |          mrmovl   varA,%eax
  0x04a: a008         |          pushl    %eax
  0x04c: 500814000000 |          mrmovl   op,%eax
  0x052: a008         |          pushl    %eax
                      |
  0x054: 805a000000   |          call     calc           # call subroutine
  0x059: 10           |          halt
                      |
  0x05a: a058         | calc:    pushl    %ebp           # prologue
  0x05c: 2045         |          rrmovl   %esp,%ebp
  0x05e: 500508000000 |          mrmovl   8(%ebp),%eax   # load op
  0x064: 308103000000 |          irmovl   $3,%ecx
  0x06a: 6110         |          subl     %ecx,%eax
  0x06c: 76b7000000   |          jg       abort
  0x071: 500508000000 |          mrmovl   8(%ebp),%eax   # load op
  0x077: 6000         |          addl     %eax,%eax      # compute 4*%eax
  0x079: 6000         |          addl     %eax,%eax
  0x07b: 308124000000 |          irmovl   table,%ecx
  0x081: 6010         |          addl     %ecx,%eax      # compute target
  0x083: 50150c000000 |          mrmovl   12(%ebp),%ecx  # load a
->0x089: 502510000000 |<-        mrmovl   16(%ebp),%edx  # load b
                      |
  0x08f: 80ba000000   |          call     trampoline
  0x094: 6021         | addop:   addl     %edx,%ecx
  0x096: 70ab000000   |          jmp      save
  0x09b: 6121         | subop:   subl     %edx,%ecx
  0x09d: 70ab000000   |          jmp      save
  0x0a2: 6221         | andop:   andl     %edx,%ecx
  0x0a4: 70ab000000   |          jmp      save
  0x0a9: 6321         | xorop:   xorl     %edx,%ecx
                      |
  0x0ab: 502514000000 | save:    mrmovl   20(%ebp),%edx
  0x0b1: 401200000000 |          rmmovl   %ecx,(%edx)
  0x0b7: b058         | abort:   popl     %ebp           # epilogue
  0x0b9: 90           |          ret
                      |
  0x0ba:              | trampoline:
  0x0ba: 500000000000 |          mrmovl   (%eax),%eax
  0x0c0: 400400000000 |          rmmovl   %eax,(%esp)
  0x0c6: 90           |          ret
                      |
 0x1000:              |          .pos 0x1000
 0x1000:              | stack:   # the stack goes here
```

Question 1

(a) What is the purpose of the subroutine `trampoline`? Why is it necessary on Y86?

(b) Explain why the `trampoline` subroutine is not required on the X86 processor? What is used instead?

Question 2

Start at address `0x000` and trace the program execution path until you reach address `0x089` (marked with `->` `<-` in the assembly listing). All CPU registers are initialised to zero before the program execution starts from address `0x000`. What is the program state *before* execution of the `mrmovl 16(%ebp),%edx` instruction?

Global Variables:

op	varA	varB	varC

Registers:

eax	ebx	ecx	edx	edi	esi	esp	ebp

Stack:

Address	Value	Description
0x0ffc		
0x0ff8		
0x0ff4		
0x0ff0		
0x0fec		
0x0fe8		

Question 3

The execution of the instruction at address 0x089 is given below. Fill out the execution of instructions at address 0x08f, 0x09b, and 0x064 similarly. Fill out both the "Generic" and the "Specific" column table which contains the effective values used during the execution.

Stage	Generic	Specific
	`mrmovl 16(%ebp),%edx`	`mrmovl 16(%ebp),%edx`
Fetch	icode:fun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_4[PC+2] valP ← PC+6	icode:fun ← M_1[0x089] = 5:0 rA:rB ← M_1[0x08a]=2:5 valC ← M_4[0x08b]=0x10 valP ← 0x089+6=0x08f
Decode	valB ← R[rB]	valB ← R[%ebp] = 0xfe8
Execute	valE ← valB + valC	valE ← 0xfe8 + 0x10 = 0xff8
Memory	valM ← M_4[valE]	valM ← M_4[0xff8] = 4
Write Back	R[rA] ← valM	R[%edx] ← 4
PC Update	PC ← valP	PC ← 0x08f

Stage	Generic	Specific
	`call trampoline`	`call 0x0ba`
Fetch		
Decode		
Execute		
Memory		
Write Back		
PC Update		

Stage	Generic	Specific
	subl %edx,%ecx	subl %edx,%ecx
Fetch		
Decode		
Execute		
Memory		
Write Back		
PC Update		

Stage	Generic	Specific
	irmovl $3,%ecx	irmovl $3,(%ecx)
Fetch		
Decode		
Execute		
Memory		
Write Back		
PC Update		

Question 4: New instruction: peek

We want to add a new instruction to the Y86 instruction set. The new instruction `peek` reads from memory at the top of the stack and stores the 32-bit value in any of the eight Y86 registers. This new instruction does not modify the stack pointer, instead it simply "peeks" at the current top of the stack. For example, the code below would put the current value found in memory at the top of the stack into register %eax.

```
peek %eax
```

The new Y86 instruction `peek` will be a two byte instruction as follows:

byte	*0*		*1*	
`peek rA`	C	0	rA	8

Fill out the following form to describe what needs to happen during each stage of this new instruction. A copy of the form for the `pushl` instruction is included for reference (just to give you a hint of what is expected). Make sure that the steps you list are possible with the *SEQ* datapath provided in the Appendix.

	pushl rA	peek rA
Fetch	icode:ifun <- M_1[PC] rA:rB <- M_1[PC+1] valP <- PC + 2	
Decode	valA <- R[rA] valB <- R[%esp]	
Execute	valE <- valB + (-4)	
Memory	M_4[valE] <- valA	
Write Back	R[%esp] <- valE	
PC Update	PC <- valP	

Question 5: New instruction: cmpl

We want to add a new instruction to the Y86 sequential implementation. The new instruction will support a comparison between two registers. This instruction would be used like the IA32 `cmpl` instruction (with the restriction that both operands are in registers). Recall that the `cmpl` instruction sets the condition codes based on the result of doing a subtraction. For example, the following instruction would subtract %edx from %eax and set the CC bits.

```
cmpl %edx, %eax
```

The new Y86 instruction `cmpl` will be a two byte instruction as follows:

byte	*0*		*1*	
cmpl rA, rB	D	0	rA	rB

In an assembly language program the registers would be listed in the same order, so in the example:

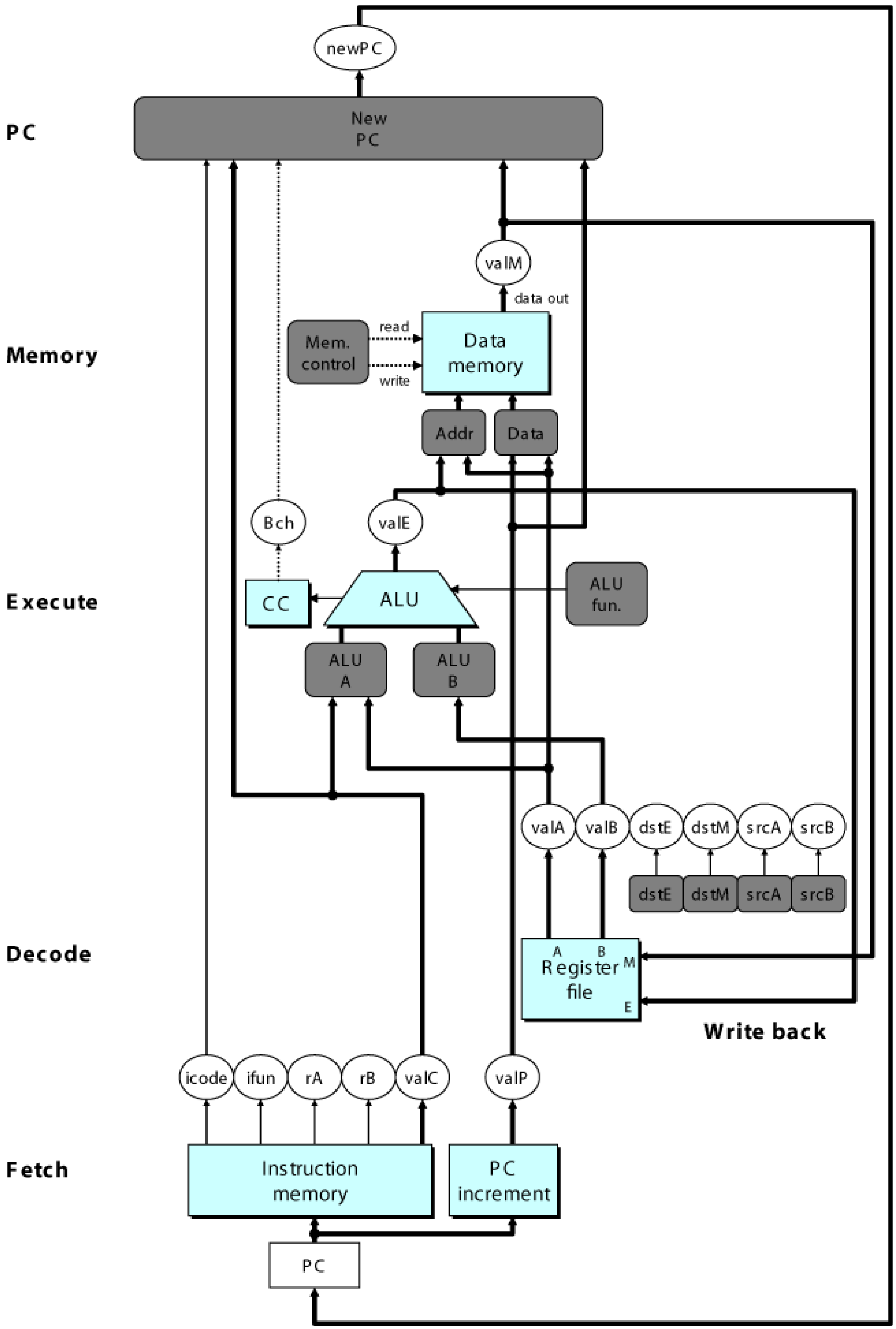
```
cmpl %edx, %eax
```

rA is %edx and *rB* is %eax. Fill out the following form to describe what needs to happen during each stage of this new instruction. A copy of the form for the `pushl` instruction is included for reference (just to remind you of what is expected). Make sure that the steps you list are possible with the *SEQ* datapath provided in the Appendix.

	pushl rA	cmpl rA, rB
Fetch	icode:ifun <- M_1[PC] rA:rB <- M_1[PC+1] valP <- PC + 2	
Decode	valA <- R[rA] valB <- R[%esp]	
Execute	valE <- valB + (-4)	
Memory	M_4[valE] <- valA	
Write Back	R[%esp] <- valE	
PC Update	PC <- valP	

Appendix A

Hardware structure of SEQ (see course book 2^{nd} edition, page 376, Figure 4.22, 1^{st} edition, page 293, Figure 4.21):



Part 2: Y86 Simulator

Install Y86 Tools

Information on installing the Y86 tools needed for this assignment can be found in the **CS:APP Guide to Y86 Processor Simulators** available on the course page (`simguide.pdf`) and during the recitation.

Question 1: Recode from X86 to Y86

Write and simulate the following Y86 programs with the sequential processor simulator (*SEQ*). The required behavior of these programs is defined by the example C functions in `examples.c`.

max.ys: Write a Y86 program (`max.ys`) that finds out the max element from a linked list. Your program should consist of a main routine that invokes a Y86 function (`max_list`) that is functionally equivalent to the C `max_list` function. You can test your program using the following three-element list:

```
  # Sample linked list
      .align 4
ele1: .long 0x00a
      .long ele2
ele2: .long 0x0b0
      .long ele3
ele3: .long 0xc00
      .long 0
```

rmax.ys: Write a recursive version of `max.ys` (`rmax.ys`) that recursively searches the max element of a linked list. Your program should consist of a main routine that invokes a recursive Y86 function (`rmax_list`) that is functionally equivalent to the C `rmax_list` function. You can test your program using the same three element list you used for testing `max.ys`.

Instructions

a) Create `max.c` based on `examples.c`

b) Generate X86 assembly code: `gcc -m32 -O1 -S max.c`

c) Rename the resulting `max.s` file to `max.ys` and recode it manually for Y86.

d) Assemble the Y86 object file: `yas max.ys`

e) Simulate the Y86 program: `ssim -g max.yo`

```c
/* examples.c */

/* linked list element */
typedef struct ELE {
   int val;
   struct ELE *next;
} *list_ptr;

/* max_list - Find out the max element of a linked list */
int max_list(list_ptr ls)
{
   int val;

   if (!ls)
      return 0;

   val = ls->val;
   ls = ls->next;
   while (ls) {
      if (val < ls->val)
         val = ls->val;
      ls = ls->next;
   }
   return val;
}

/* rmax_list - Recursive version of max_list */
int rmax_list(list_ptr ls)
{
   int val, max_rest;

   if (!ls)
      return 0;

   val = ls->val;
   if (ls->next){
      max_rest = rmax_list(ls->next);
      if (val< max_rest)
         return max_rest;
   }
   return val;
}
```

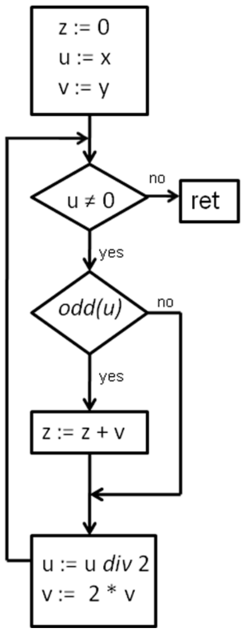
Question 2: Multiplication for Y86

We want to perform multiplication of two unsigned numbers in a Y86 program. Since our simulator does not provide a multiplication instruction, we have to implement it by hand.

The flowchart below illustrates an algorithm to multiply two numbers `x` and `y`. The algorithm corresponds to manual multiplication learned in school, but works on binary instead of decimal numbers.

Your task is to complete the `Multiply` function in the given Y86 program skeleton with the illustrated multiplication algorithm. You should do this by hand.

Flowchart:



Template:

```
# Execution begins at address 0
      .pos      0
Init:
      irmovl    Stack, %esp     # set up stack pointer
      irmovl    Stack, %ebp     # set up base pointer
      jmp       Main

Multiply:
#
# fill out the body
# put result in %eax
#
      ret

Main:
      irmovl  $3, %esi          # push y
      pushl   %esi
      irmovl  $5, %esi          # push x
      pushl   %esi
      call    Multiply          # multiply(x, y)
      halt

      .pos    0x100
Stack:                          # the stack goes here
```

Instructions

a) Fill in the missing function `multiply(x,y)`.

b) Assemble the Y86 object file: `yas multiply.ys`

c) Simulate the Y86 program: `ssim -g multiply.yo`

Hints:

- You don't need to program a function *odd(u)* — how can you simply check for an odd number?
- Since Y86 does not support division or shift operations, the algorithm needs to be slightly modified from the version in the flowchart. Consider how bit-masks with a single bit set to `1` might help you here.

Hand in solutions

Hand in the paper exercises to your assistant during the recitation sessions and hand in the simulator exercises via SVN, in the same way as the previous programming assignments.

Hints

The student lab machines (stud[1..26]-h[56..57].inf.ethz.ch) have the necessary packages installed to compile the simulator. If you choose to use your own Linux machine, you will need to install at least the following packages (ask your TA if you have more questions):

- flex
- tcl-dev
- tk-dev
- bison