

Computer Organization

Spring 2005

Final Exam

Name Sample Answers

There are 15 pages - make sure you have all of them.
Answer all questions - pay attention to the # of points for each question.
Don't leave anything blank - partial credit is always possible!

Question 1 (5 pts) : Write Y86 Assembly code that provides the same functionality as the following IA32 instructions. The values in `%ecx`, `%ebp` and `%esi` are established before the code starts; you can use any other registers to hold intermediate values. **IMPORTANT**: your code must not change `%ecx`, `%ebp` or `%esi` (since the IA32 instructions don't!).

IA32

```
addl 25(%esi,%ecx,2),%eax
addl 8(%ebp),%eax
```

Y86

```
rrmovl %ecx,%edx
addl %edx,%edx # edx = ecx*2
addl %esi,%edx # edx = ecx*2+eci
movl 25(%edx),%ebx # 25(%eci,%ecx,2)-> ebx
addl %ebx,%eax # eax+=25(%eci,%ecx,2)

movl 8(%ebp),%edx # edx = 8(%ebp)
addl %edx,%eax # eax = 8(%ebp) + eax
```

Question 2 (5 pts): We studied two different techniques for timing programs or parts of programs. Briefly describe how interval counting and cycle counting work.

Interval counting is a statistical system, at regular intervals (a few ms.) a record is made of what process/function is running. The process/function is credited with having the CPU for the entire interval. This will generally be inaccurate if done for just a few time intervals, but works well for measurements taking thousands of intervals or more.

Cycle counting is based on having a hardware cycle counter available that is automatically incremented each cycle (typically we need a large counter to avoid wrap-around). A program can read the cycle counter before and after doing some operations, then subtract the two to get the elapsed number of cycles. This will not be accurate if the process was not running on the CPU during the timing, and in general the process has no way of accounting for this. Since this scheme will never underestimate the number of cycles it takes to do the operations (but will often overestimate), a k-best strategy can be used.

Question 3: A virtual memory system is based on 32 bit virtual addresses. Each page is 2k bytes (2048 bytes). A Translation Look-aside Buffer is used to cache page table entries, the TLB is direct mapped and holds a total of 2k page table entries.

(2pts) How many of the bits from a virtual address are used to determine where in the page table to look for the physical page number?

2^{32} bytes / 2^{11} bytes/page $\rightarrow 2^{21}$ pages, each has a unique page address, so the page addresses are 21 bits long.

(2pts) How many (potential) entries are there in each pagetable (each process has it's own pagetable)?

Each pagetable could have 2^{21} page table entries (2M page table entries)

(3 pts) Assume the TLB is currently full. How many unique bytes of memory (how many different byte addresses) can be accessed without needing to go to memory for the address translation?

TLB holds 2^{11} (2K) page table entries, each entry is the translation for one page (2^{11} (2K) bytes). Total memory addressable without going to the page table is:

$$2^{11} * 2^{11} = 2^{22} (4M) \text{ bytes}$$

Question 4: The following problem concerns basic cache lookups.

- The memory is byte addressable.
- Memory accesses are to bytes.
- Physical addresses are 14 bits wide.
- The cache is direct-mapped, with a 8 byte block size and 8 slots.

The cache contents are shown, all numbers are hex.

Direct Mapped Cache

Slot #	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
0	09	1	86	30	3F	10	AE	2D	34	71
1	25	1	60	4F	E0	23	8D	A0	2F	33
2	5D	1	2F	81	FD	09	02	0B	FF	FE
3	06	0	3D	94	9B	F7	4E	17	00	01
4	14	1	06	78	07	C5	22	81	14	44
5	35	0	0B	DE	18	4B	05	20	57	10
6	0B	1	A0	B7	26	2D	28	D3	1F	CC
7	33	0	B1	0A	32	0F	8F	BA	C9	B8

(3 pts) For the address $0x1755$, what slot will be checked?

address looks like this:

Slot is 010 (slot 2)

Tag								Slot		Offset	
0	1	0	1	1	1	0	1	0	1	0	1

(3 pts) Is address $0x1755$ a cache hit or miss? If a hit, what is the byte value is returned (what does the cache say is the byte value for address $0x1755$).

Tag is 01011101 which is 0x5D which is currently in slot 2, so it's a hit.

Byte returned depends on the offset, which is 101, so byte 5 is returned.

value of byte 5 in slot 2 is **0B**

Question 5: We want to add a new instruction to the Y86 instruction set that reads from memory at the top of the stack and stores the value in any register. This new instruction does not modify the stack pointer, instead it simply *peeks* at the current top of the stack. Our new instruction is named `peek`, this new instruction will result in putting the 32 bit value currently stored at the top of the stack into any of the 8 Y86 registers. For example, the code below would put the current value found in memory at the top of the stack into register `%eax`.

`peek %eax`

(8 pts): The new Y86 instruction will be a two byte instruction as follows:

icode:ifun

E	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 push instruction is included for reference (just to remind you of what kind of stuff is expected). Make sure that the steps you list are possible with the datapath shown in the modified Fig 4.21 (one of the handouts supplied with this test).

	pushl rA	peek
Fetch	<code>icode:ifun <- M₁[PC] rA:rB <- M₁[PC+1] valP <- PC+2</code>	<code>icode:ifun <- M₁[PC] rA:rB <- M₁[PC+1]</code>
Decode	<code>valA <- R[rA] valB <- R[%esp]</code>	<code>valA <- R[%esp]</code>
Execute	<code>valE <- valB + (-4)</code>	<code>valE <- valA + 0</code>
Memory	<code>M₄[valE] <- valA</code>	<code>valM <- M₄[valE]</code>
Write Back	<code>R[%esp] <- valE</code>	<code>R[rA] <- valM</code>
PC update	<code>PC <- valP</code>	<code>PC <- valP</code>

Question 5 continued. (7 pts): Modify the HCL expressions show below to support your peek instruction. You can assume that the symbol IPEEK is defined to represent the icode for this new instruction. Your HCL must reflect the actions you listed in your table (from part a) to be considered correct! The HCL listed below is not complete (some controls are not shown) – just modify the ones listed below, don't worry about any that are not listed.

```

# Does fetched instruction require a regid byte?
bool need_regids =
    icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
              IIRMOVL, IRMMOVL, IMRMOVL, IPEEK };

# Does fetched instruction require a constant word?
bool need_valC =
    icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };

## What register should be used as the A source?
int srcA = [
    icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : rA;
    icode in { IPOPL, IRET, IPEEK } : RESP;
    1 : RNONE; # Don't need register
];

## What register should be used as the B source?
int srcB = [
    icode in { IOPL, IRMMOVL, IMRMOVL } : rB;
    icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
    1 : RNONE; # Don't need register
];

## What register should be used as the E destination?
int dstE = [
    icode in { IRRMOVL, IIRMOVL, IOPL } : rB;
    icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
    1 : RNONE; # Don't need register
];

## What register should be used as the M destination?
int dstM = [
    icode in { IMRMOVL, IPOPL, IPEEK } : rA;
    1 : RNONE; # Don't need register
];

## Select input A to ALU
int aluA = [
    icode in { IRRMOVL, IOPL } : valA;
    icode in { IIRMOVL, IRMMOVL, IMRMOVL } : valC;
    icode in { ICALL, IPUSHL } : -4;
    icode in { IRET, IPOPL } : 4;
    # Other instructions don't need ALU
];

## Select input B to ALU
int aluB = [
    icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
              IPUSHL, IRET, IPOPL } : valB;
    icode in { IRRMOVL, IIRMOVL } : 0;
    # Other instructions don't need ALU
];

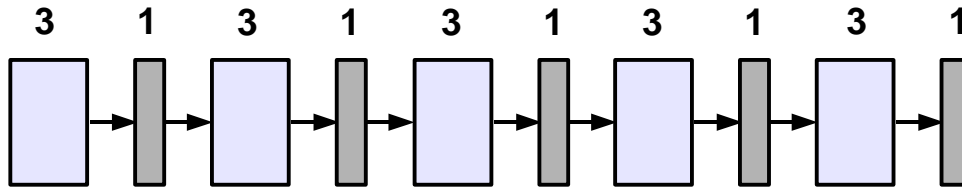
## Set read control signal
bool mem_read = icode in { IMRMOVL, IPOPL, IRET, IPEEK };

## Set write control signal
bool mem_write = icode in { IRMMOVL, IPUSHL, ICALL };

## Select memory address
int mem_addr = [
    icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
    icode in { IPOPL, IRET, IPEEK } : valA;
    # Other instructions don't need address
];

```

Question 6a (3 pts): A 5 stage pipelined processor (pictured below) has maximum throughput of N instructions/second. Each stage requires 3 units of time for the combinational circuit and 1 unit for the pipeline registers.

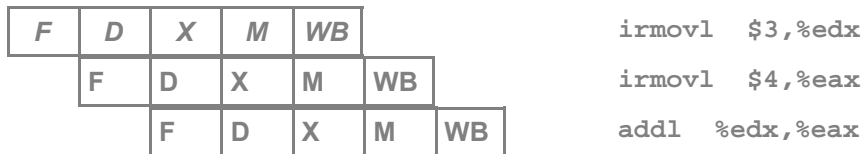


Assume that we combine all the combinational circuits into one big combinational circuit. What is the throughput of the resulting sequential implementation in terms of N ?

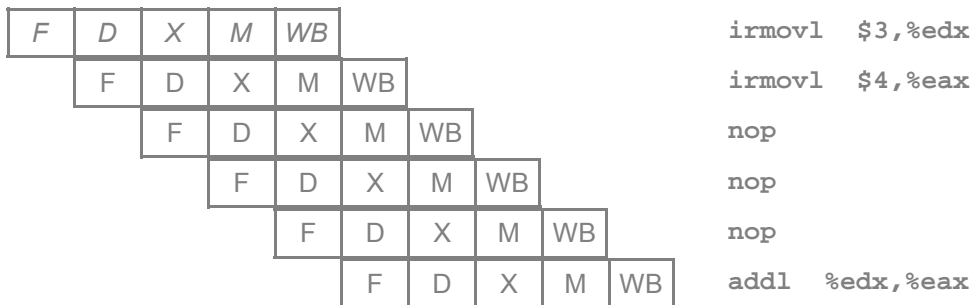
Pipelined version completes one instruction every 4 units of time.
 Sequential version completes one instruction every 16 units of time
 (only 1 register is needed, so time is $5 \cdot 3 + 1$)
 Sequential has 1/4 th of the throughput: $N/4$ instructions/second.

Question 6b (3 pts): There is a data hazard in the following Y86 assembly code. Briefly describe the hazard and rewrite the code with just enough NOPS inserted to eliminate the problem (no extra NOPS!). You should assume the processor is based on the 5 stage Y86 pipeline we discussed in class (stages Fetch, Decode, Execute, Memory, Writeback).

```
irmovl $3,%edx
irmovl $4,%eax
addl %edx,%eax
```



Decode stage of the add instruction will read `%edx` and `%eax`, but neither has the correct value yet. `edx` will be correct after the WB of the first instruction, `eax` will be correct one cycle later. We need to make sure the D stage of the add instruction doesn't execute until both of the other instructions have completed, we need to insert 3 NOPS:



Question 7 (20 pts) Fill in the blanks, 2 points each

- How many bytes can be addressed on a computer with 26 bit addresses? (assume a single, linear, byte addressable address space). 2²⁶ = 64MBytes
- reading a word from a hard disk involves three time consuming parts:
 - seek
 - rotational delay
 - transferwhich is the least time consuming (which is the fastest)? transfer
- What is the representation of the 8 bit twos-complement number -33? 1101 1111 or 0xDF
- What is the maximum throughput in instructions/cycle for a 5 stage pipeline? 1 instruction/cycle
- Show a single IA32 instruction puts a zero on the top of the stack, but does not change the stack pointer. movl \$0,(%esp)
- Using K-best strategy with K=3 and tolerance 1, an initial set of measurements (cycle counts) is taken for a code sequence. The cycle counts are shown below. What is the reported run time in cycles ?
22, 24, 18, 21, 21, 19, 18, 21, 20 18
- What is the name of a common combinational circuit that is used to select from a number of inputs? Multiplexer
- What value will be in %eax after these instructions:
push \$1234
call foo
foo: popl %eax foo
- What is the value of this C expression (c and x are unsigned char):
(c ^ x) ^ c x
- Is the IA32 instruction set RISC or CISC? CISC

Question 8 (5 pts): Consider the following C code that does a 32 bit unsigned multiplication by repeated addition. Rewrite the code so that it will run faster, and include a one line description of each optimization you make to the code. You are not allowed to simply use multiplication operations (it should still multiply using repeated addition, and should still return 0 if the resulting product cannot be represented as a 32 bit unsigned int).

```

/* returns true if adding x to curtotal will result in overflow */
int will_overflow(unsigned int curtotal, unsigned int x) {
    if (curtotal + x < curtotal)
        return(1);
    else
        return(0);
}

/* unsigned multiplication by repeated addition */
unsigned int multiply(unsigned int x, unsigned int y) {
    unsigned int i=0;
    unsigned total=0;

    while ( (i<y) && (! will_overflow(total,x))) {
        total = total + x;
        i++;
    }

    /* check if we have finished or were stopped by an overflow */
    if (i<y) {
        /* error - would overflow! */
        return(0);
    } else {
        return(total);
    }
}

```

Potential Optimizations:

1. inlining the code for will_overflow (moving into the multiply function). Eliminates the overhead of one function call per loop iteration.

2. loop unrolling. Could precompute x+x outside the loop, then add this value $\frac{1}{2}$ as many times. Could also precompute x+x+x+x, and do $\frac{1}{4}$ the number of loops, etc.

3. could get rid of i by having y count down, this could keep the counter in a register.

```

/* unsigned multiplication by repeated addition */
unsigned int multiply(unsigned int x, unsigned int y) {
    unsigned total=0;
    unsigned twox = x+x;

    while ( (y>1) && (total+twox>total)) {
        total = total + twox;
        y-=2;
    }
    if ((y==1)&&(total+x>total)) {
        total+=x; y=0;
    }

    if (y) {
        /* error - would overflow! */
        return(0);
    } else {
        return(total);
    }
}

```

Question 9 (4 pts) Consider the following C functions and assembly code:

```
int fun4(int *ap, int *bp) {
    int a = *ap;
    int b = *bp;
    return a+b;
}

int fun5(int *ap, int *bp) {
    int b = *bp;
    *bp += *ap;
    return b;
}

int fun6(int *ap, int *bp) {
    int a = *ap;
    *bp += *ap;
    return a;
}
```

```
pushl %ebp
movl %esp,%ebp
movl 8(%ebp),%edx
movl 12(%ebp),%eax
movl (%edx),%edx
addl %edx,(%eax)
movl %edx,%eax
popl %ebp
ret
```

Which of the functions compiled into the assembly code shown?

Code is from fun6

```
pushl %ebp
movl %esp,%ebp
movl 8(%ebp),%edx           # edx is ap
movl 12(%ebp),%eax         # eax is bp
movl (%edx),%edx           # edx is *ap
addl %edx,(%eax)           # *bp += *ap
movl %edx,%eax             # put *ap in eax (this is a)
popl %ebp
ret
```

Question 10: Short answer – 4 points each.

- Describe (the specific problem) what goes wrong if I call `palindrome("abcdefghijklkkjihgfedcba");`.

```
int reverse(char *dest, char *src) {
    dest += strlen(src);
    *dest=0;
    dest--;
    while (*src) {
        *dest = *src;
        dest--; src++;
    }
}

int palindrome(char *s) {
    char buf[10];
    reverse(buf,s);
    if (strcmp(buf,s)==0) {
        return(1);
    } else {
        return(0);
    }
}
```

The return address of the function `palindrome` will be clobbered when `reverse` creates a reversed copy of the string in `buf`. In this case `palindrome` will try to return to an address something like “fedc”, and will crash.

Increasing block size directly supports spatial locality, since we move more surrounding memory elements into the cache (along with the referenced element). lity.

Increasing the block size (alone), will decrease the number of cache slots, this will be bad for temporal locality, since we can't hold as many recently referenced memory elements (but can hold more neighbors).

- Briefly explain what a normalized binary floating point number is, and state why normalized numbers are used. Normalized means that we adjust the exponent until there is a single 1 immediately to the left of the binary point.

By insisting on using normalized numbers, the IEEE format saves one bit of storage for the mantissa – we know the first bit is a 1 so there is no need to store it.

Question 11: Consider a direct mapped cache of size 1K bytes. You will calculate the miss rate for the following code using this cache. Remember that `sizeof(int) == 4`. Assume that the cache starts empty and that local variables and computations take place completely within the registers (only the matrix is in memory).

```
int sum_nth_array_elements_twice(int m[128],int n){
    int i;
    int tot=0;

    for (i=0; i<128; i+=n) {
        tot+= m[i];
    }
    for (i=0; i<128; i+=n) {
        tot+= m[i];
    }
    return(tot);
}
```

(3 pts) What is the cache miss rate if block size is 4 bytes and n is 1?

Miss rate = 50 %

(3pts) What is the cache miss rate if the block size is 16 bytes and n is 2?

Miss rate = 75 %

The cache can hold the array (array requires $128 \times 4 = 512$ bytes, cache holds twice that). This means the second loop will always be 100% hits.

For block size 4 bytes, each access is to one int, and the first run through the array will be 100% misses (no support for spatial locality here...). Total miss rate over both loops will be 50%.

For block size 16 bytes and $n=2$, accessing the first array element will bring along the next 3 as well, but we skip over two of them. This means the first loop will have a miss rate of 50%. So total miss rate is 25% (for both loops)

Question 12 (9pts) Consider the following incomplete definition of a C struct along with the incomplete code for a function func given below in C.

```

typedef struct node {
    double x;
    unsigned int y;
    struct node *next;
    struct node *prev;
} node_t;

node_t n;

void func() {
    node_t *m;
    m = n.next->prev;
    m->y /= 16;
    return;
}

```



```

func:
    pushl %ebp
    movl %esp,%ebp
    movl n+12,%eax
    movl 16(%eax),%eax
    shrw $4,8(%eax)
    leave
    ret

```

When the C code was compiled on an IA-32 machine running Linux, the assembly code shown was generated for function func.

Given these code fragments, fill in the blanks in the C code given above. **Note: there is a unique answer.** The types must be chosen from the following table, assuming the sizes and alignment given.

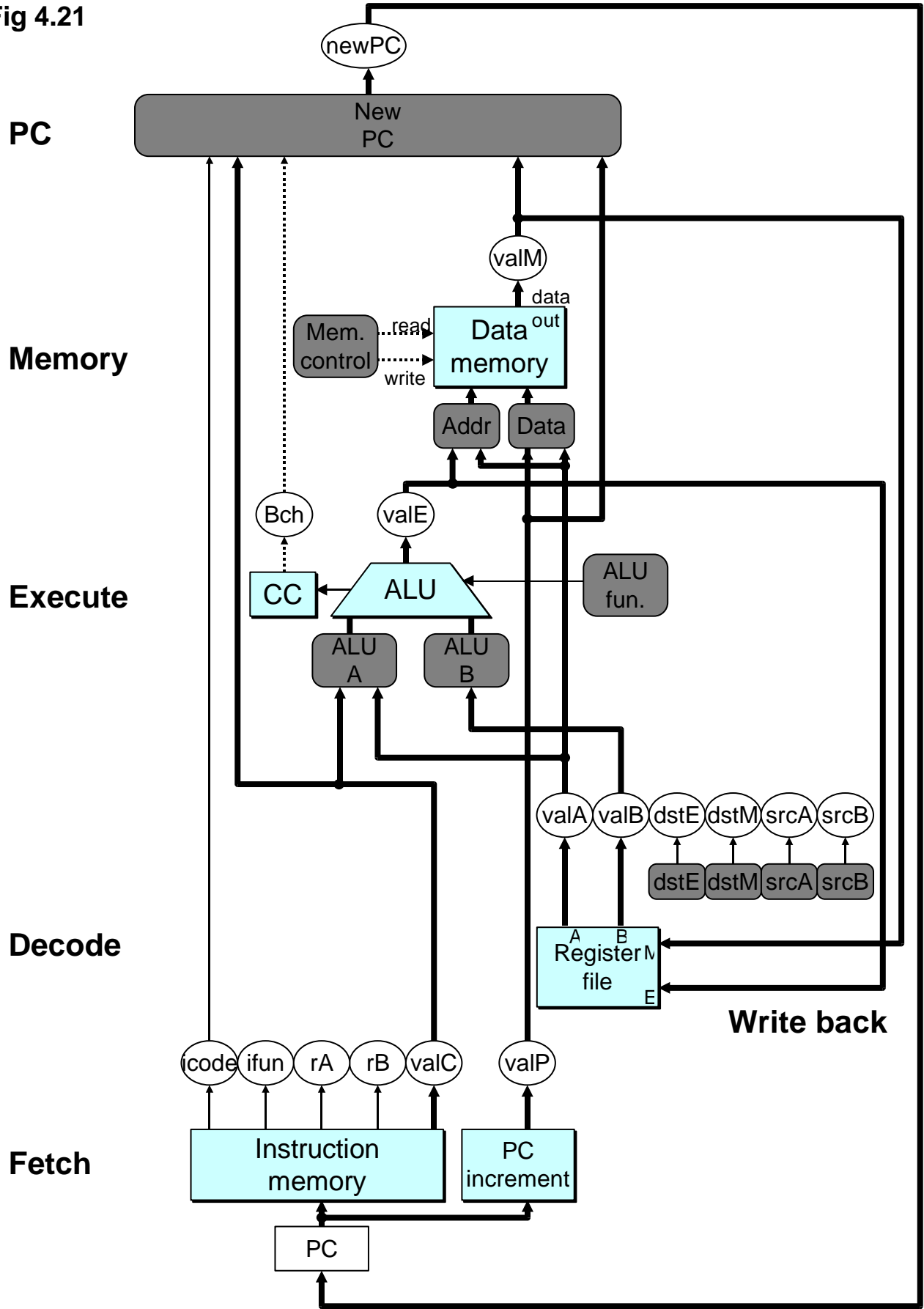
Type	Size (bytes)	Alignment (bytes)
char	1	1
short	2	2
unsigned short	2	2
int	4	4
unsigned int	4	4
double	8	4

We can tell from the shift instruction that the offset to y is 8, this tells us that x must use 8 bytes, the only possibility is a double.

From the shift we can tell that y is an unsigned int (logical shift), this means that n+12 is next.

Offset 16 is prev, so we can now determine that the assignment to m is n.next->prev.

Fig 4.21



Some IA32 Instructions

<code>movl Src, Dest</code>	<i>Dest = Src</i>
<code>addl Src, Dest</code>	<i>Dest = Dest + Src</i>
<code>subl Src, Dest</code>	<i>Dest = Dest - Src</i>
<code>imull Src, Dest</code>	<i>Dest = Dest * Src</i>
<code>sall Src, Dest</code>	<i>Dest = Dest << Src</i>
<code>sarl Src, Dest</code>	<i>Dest = Dest >> Src</i>
<code>shrl Src, Dest</code>	<i>Dest = Dest >> Src</i>
<code>xorl Src, Dest</code>	<i>Dest = Dest ^ Src</i>
<code>andl Src, Dest</code>	<i>Dest = Dest & Src</i>
<code>orl Src, Dest</code>	<i>Dest = Dest Src</i>
<code>incl Dest</code>	<i>Dest = Dest + 1</i>
<code>decl Dest</code>	<i>Dest = Dest - 1</i>
<code>negl Dest</code>	<i>Dest = - Dest</i>
<code>notl Dest</code>	<i>Dest = ~ Dest</i>
<code>leal Src, Dest</code>	<i>Dest = address of Src</i>
<code>cmpl Src2, Src1</code>	<i>Sets CCs Src1 Src2</i>
<code>testl Src2, Src1</code>	<i>Sets CCs Src1 & Src2</i>

<code>jmp label</code>	<i>jump</i>
<code>je label</code>	<i>jump equal</i>
<code>jne label</code>	<i>jump not equal</i>
<code>js label</code>	<i>jump negative</i>
<code>jns label</code>	<i>jump non-negative</i>
<code>jg label</code>	<i>jump greater (signed)</i>
<code>jge label</code>	<i>jump greater or equal (signed)</i>
<code>jl label</code>	<i>jump less (signed)</i>
<code>jle label</code>	<i>jump less or equal (signed)</i>
<code>ja label</code>	<i>jump above (unsigned)</i>
<code>jb label</code>	<i>jump below (unsigned)</i>
<code>push Src</code>	<i>Mem[%esp-4] = Src, %esp=%esp-4</i>
<code>pop Dest</code>	<i>Dest = Mem[%esp], %esp=%esp+4</i>
<code>call label</code>	<i>Mem[%esp-4] = %eip, %esp=%esp-4, %eip = label</i>
<code>ret</code>	<i>%eip = mem[%esp], %esp=%esp+4</i>

IA32 Addressing Modes

Immediate	\$val	value
Normal	(R)	Mem[Reg[R]] -Register R specifies memory address <code>movl (%ecx), %eax</code>
Displacement	D(R)	Mem[Reg[R]+D] -Register R specifies start of memory region -Constant displacement D specifies offset <code>movl 8(%ebp), %edx</code>
Indexed	D(Rb,Ri,S)	Mem[Reg[Rb]+S*Reg[Ri]+ D] -D: Constant "displacement" 1, 2, or 4 bytes -Rb: Base register: Any of 8 integer registers -Ri: Index register: -Scale: 1, 2, 4, or 8 <code>movl -4(%ebx, %esi, 4), %edx</code>

IA32 Registers

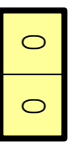
<code>%eax</code>	<code>%ebx</code>	<code>%ecx</code>	<code>%edx</code>	<code>%esi</code>	<code>%edi</code>	<code>%ebp</code>	<code>%esp</code>
-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------

Y86 Instruction Set

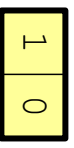
Byte

0 1 2 3 4 5

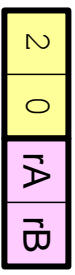
nop



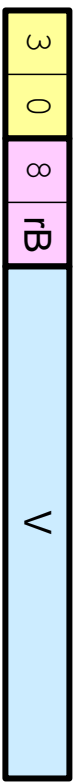
halt



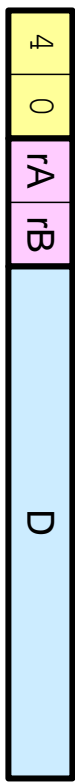
rmmovl rA, rB



lrmovl V, rB



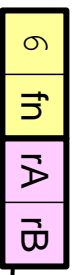
rmmovl rA, D(rB)



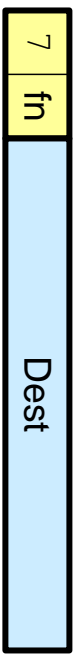
mrmovl D(rB), rA



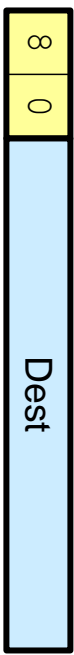
opl rA, rB



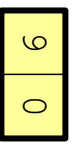
jXX Dest



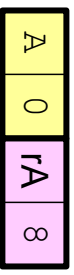
call Dest



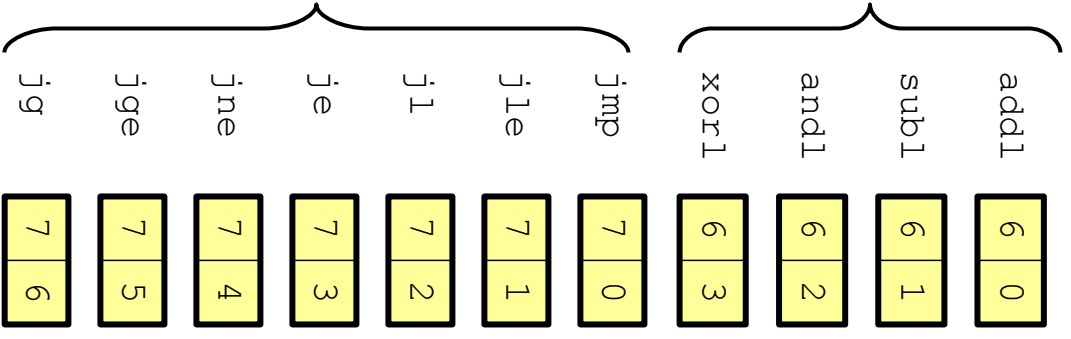
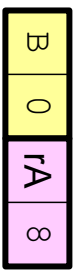
ret



pushl rA



popl rA



HCL Operations

Boolean Expressions

Logic Operations: $a \ \&\& \ b, a \ \|\| \ b, !a$

Word Comparisons: $A == B, A != B, A < B, A <= B,$
 $A >= B, A > B$

Set Membership: $A \text{ in } \{ B, C, D \}$

Same as $A == B \ \|\| \ A == C \ \|\| \ A == D$

Word Expressions

Case expressions $[a : A; b : B; c : C]$

Evaluate test expressions a, b, c, \dots in sequence

Return word expression A, B, C, \dots for first successful test

Question	Possible	Score
1	5	
2	5	
3	7	
4	6	
5	15	
6	6	
7	20	
8	5	
9	4	
10	12	
11	6	
12	9	
Total	100	