IT5002 Computer Organization
Tutorial 1
C and Number Systems
SUGGESTED SOLUTIONS

1. In 2's complement representation, "sign extension" is used when we want to represent an *n-bit* signed integer as an *m-bit* signed integer, where $m > n$. We do this by copying the sign-bit of the *n-bit* signed $m - n$ times to the left of the *n-bit* number to create an *m*-bit number.

   So for example, we want to sign-extend 0b0110 to an 8-bit number. Here $n = 4$, $m = 8$, and thus we copy the sign but $m - n = 4$ times, giving us 0b00000110.

   Similarly if we want to sign-extend 0b1010 to an 8-bit number, we would get 0b11111010.

   Show that IN GENERAL sign extension is value-preserving. For example, 0b00000110 = 0b0110 and 0b11111010 = 0b1010.

Answer:

If the sign bit is zero, this is straightforward, since padding more 0's to the left add nothing to the final value. If the sign bit is one, this is trickier to prove. In the original *n-bit* representation, the leftmost bit has a weight of -2^{n-1} giving us $X = -2^{n-1} + b_{n-2} . 2^{n-2} + b_{n-3} . 2^{n-3} + b_0$.

Let $Z = b_{n-2} . 2^{n-2} + b_{n-3} . 2^{n-3} + b_0$, then $X = -2^{n-1} + Z$

In the new m bit representation where $m > n$, the leftmost bit has a weight of -2^{m-1}, and since we copy the leftmost bit (i.e. the leftmost bit) a total of $m - n$ times. We get $Y = -2^{m-1} + 2^{m-2} + 2^{m-3} + \dots + 2^n + 2^{n-1} + Z$.

We also see that $-2^{m-1} + 2^{m-2} = -2^{m-1} + 2^{-1}.2^{m-1}$
$= 2^{m-1}(-2^0 + 2^{-1})$
$= 2^{m-1} . -2^{-1}$
$= -2^{m-2}$.

Hence in general, $-2^{m-1} + 2^{m-2} = -2^{m-2}$, and from this we have:
$-2^{m-2} + 2^{m-3} = -2^{m-3}$
$-2^{m-3} + 2^{m-4} = -2^{m-4}$
…
$-2^n + 2^{n-1} = -2^{n-1}$

Thus $Y = -2^{m-1} + 2^{m-2} + 2^{m-3} + \dots + 2^n + 2^{n-1} + Z$
$= -2^{n-1} + Z$
$= X$

2. We generalize (r – 1)'s-complement (also called radix diminished complement) to include fraction as follows:

$$\mathbf{(r - 1)\text{'s complement of } N = r^n - r^{-m} - N}$$

where n is the number of integer digits and m the number of fractional digits. (If there are no fractional digits, then m = 0 and the formula becomes r^n – 1 – N as given in class.)

For example, the 1's complement of 011.01 is (2^3 – 2^{-2}) – 011.01 = (1000 – 0.01) – 011.01 = 111.11 – 011.01 = 100.10.

Perform the following binary subtractions of values represented in 1's complement representation by using addition instead. (Note: Recall that when dealing with complement representations, the two operands must have the same number of digits.)

Is sign extension used in your working? If so, highlight it.

Check your answers by converting the operands and answers to their actual decimal values.

(a) 0101.11 – 010.0101
(b) 010111.101 – 0111010.11

Answers:

(a) 0101.1100 – 0010.0101 → 0101.1100 + 1101.1010 → **0011.0111_{1s}**
(Check: 5.75 – 2.3125 = 3.4375)

(b) 0010111.101 – 0111010.110 → 0010111.101 + 100010
1011100.110_{1s} = -0100011.001_2
(Check: 23.625 – 58.75 = –35.125)

Note that sign-extension is used above.

Note that two trailing zeroes are added. (This is not sign extension.)

3. Convert the following numbers to fixed-point binary in 2's complement, with 4 bits for the integer portion and 3 bits for the fraction portion:

(a) 1.75
$(0001.110)_{2s}$

(b) -2.5
We begin with 2.5: $(0010.100)_{2s}$
Invert and +0.001: $(1101.100)_{2s}$

(c) 3.876
0.876 * 2 = 1.752
0.752 * 2 = 1.504
0.504 * 2 = 1.008

0.008 * 2 = 0.016

So 0.876 = 0.111_{2s}
Our number is $(0011.111)_{2s}$

(d) 2.1
0.1 * 2 = 0.2
0.2 * 2 = 0.4
0.4 * 2 = 0.8
0.8 * 2 = 1.6

So 0.1 = 0.0001_{2s} = 0.001_{2s}

Putting it together gives us:
2.1 = $(0010.001)_{2s}$

Using the binary representations you have just derived, convert them back into decimal. Comment on the compromise between range and accuracy of the fixed-point binary system.

The first two will convert back exactly to 1.75 and -2.5, so that's ok.

For c:

The fraction part is 0.111_2 = 0.5 + 0.25 + 0.125 = 0.875, which is just off the target of 0.876 by 0.001. Not bad.

For d:

The fraction part is 0.001_2 = 0.125. This is off the actual value of 0.1 by 0.025, quite a lot.

Comment: Not all numbers can be represented exactly, and the precision depends on the number of bits in the fraction part. In this case 3 bits is too little to even represent 0.1, because the smallest fraction it can represent is 0.125.

4. [AY2010/2011 Semester 2 Term Test #1]
   How would you represent the decimal value –0.078125 in the IEEE 754 single-precision representation? Express your answer in hexadecimal. Show your working.

Answer: **B D A 0 0 0 0 0**

$-0.078125 = -0.000101_2 = -1.01 \times 2^{-4}$

Exponent = $-4 + 127 = 123 = 01111011_2$

1   01111011   0100000…

1011   1101   1010   0000   …

B D A 0 0 0 0 0

5. Given the partial C program shown below, complete the two functions: **readArray()** to read data into an integer array (with at most 10 elements) and **reverseArray()** to reverse the array. For **reverseArray()**, you are to provide two versions: an iterative version and a recursive version. For the recursive version, you may write an auxiliary/driver function to call the recursive function.

```
#include <stdio.h>
#define MAX 10

int readArray(int [], int);
void printArray(int [], int);
void reverseArray(int [], int);

int main(void) {
   int array[MAX], numElements;

   numElements = readArray(array, MAX);
   reverseArray(array, numElements);
   printArray(array, numElements);

   return 0;
}

int readArray(int arr[], int limit) {

   // ...
   printf("Enter up to %d integers, terminating with a negative
integer.\n", limit);
   // ...
}

void reverseArray(int arr[], int size) {

   // ...
}

void printArray(int arr[], int size) {
   int i;
   for (i=0; i<size; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");
}
```

Answers:

```
int readArray(int arr[], int limit) {
   int i, input;

   printf("Enter up to %d integers, terminating with a negative
integer.\n", limit);
   i = 0;
   scanf("%d", &input);
   while (input >= 0) {
      arr[i] = input;
      i++;
      scanf("%d", &input);
   }
   return i;
}
```

```
// Iterative version
// Other solutions possible
void reverseArray(int arr[], int size) {
   int left=0, right=size-1, temp;

   while (left < right) {
      temp = arr[left]; arr[left] = arr[right]; arr[right] = temp;
      left++; right--;
   }
}
```

```
// Recursive version
// Auxiliary/driver function for the recursive function
// reverseArrayRec()
void reverseArrayV2(int arr[], int size) {
   reverseArrayRec(arr, 0, size-1);
}

void reverseArrayRec(int arr[], int left, int right) {
   int temp;

   if (left < right) {
      temp = arr[left]; arr[left] = arr[right]; arr[right] = temp;
      reverseArrayRec(arr, left+1, right-1);
   }
}
```

6. Trace the following program manually (do not run it on a computer) and write out its output. When you present your solution, draw diagrams to explain.

```
#include <stdio.h>

int main(void) {
   int a = 3, *b, c, *d, e, *f;

   b = &a;
   *b = 5;
   c = *b * 3;
   d = b;
   e = *b + c;
   *d = c + e;
   f = &e;
   a = *f + *b;
   *f = *d - *b;

   printf("a = %d, c = %d, e = %d\n", a, c, e);
   printf("*b = %d, *d = %d, *f = %d\n", *b, *d, *f);

   return 0;
}
```

Remember to post on the LumiNUS forum if you have any queries.

Answers:

```
a = 55, c = 15, e = 0
*b = 55, *d = 55, *f = 0
```

IT5002 Computer Organization
Tutorial 2
ANSWERS

Tutorial Questions:

1. Below is a C code that performs palindrome checking. A palindrome is a sequence of characters that reads the same backward or forward. For example, “madam” and “rotator” are palindromes.

```
char string[size] = { ... }; // some string
int low, high, matched;

// Translate to MIPS from this point onwards
low = 0;
high = size-1;
matched = 1;          // assume this is a palindrome
                      // In C, 1 means true and 0 means false
while ((low < high) && matched) {
   if (string[low] != string[high])
      matched = 0;    // found a mismatch
   else {
      low++;
      high--;
   }
}
// "matched" = 1 (palindrome) or 0 (not palindrome)
```

Given the following variable mappings:

low ➔ $s0;
high➔ $s1;
matched ➔ $s3;
base address of string[] ➔ $s4;
size ➔ $s5

a. Translate the C code into MIPS code by keeping track of the indices.

b. Translate the C code into MIPS code by using the idea of “array pointer”. Basically, we keep track of the actual addresses of the elements to be accessed, rather than the indices. Refer to lecture set #8, slide 34 for an example.

Note: Recall the “short circuit” logical AND operation in C. Given condition (A && B), condition B will not be checked if A is found to be false.

To tutors: It may help if you can project the programs in Q1-3 on the whiteboard. For Q2, you may also get students to write their answers overlay with the projected program on the whiteboard.
For room with a lot of whiteboards, you may consider getting students to write their answers for different on the whiteboards concurrently to save time.

Answers:

a.

```
       addi $s0, $zero, 0     # low = 0
       addi $s1, $s5, -1      # high = size-1
       addi $s3, $zero, 1     # matched = 1
loop:  slt  $t0, $s0, $s1     # (low < high)?
       beq  $t0, $zero, exit  # exit if (low >= high)
       beq  $s3, $zero, exit  # exit if (matched == 0)
       add  $t1, $s4, $s0     # address of string[low]
       lb   $t2, 0($t1)       # t2 = string[low]
       add  $t3, $s4, $s1     # address of string[high]
       lb   $t4, 0($t3)       # t4 = string[high]
       beq  $t2, $t4, else
       addi $s3, $zero, 0     # matched = 0
       j endW                 # can be "j loop"
else:  addi $s0, $s0, 1       # low++
       addi $s1, $s1, -1      # high—
endW:  j loop                 # end of while
exit:                         # outside of while
```

b.

```
       addi $s0, $zero, 0     # low = 0
       addi $s1, $s5, -1      # high = size-1
       addi $s3, $zero, 1     # matched = 1
       add  $t1, $s4, $s0     # address of string[low]
       add  $t3, $s4, $s1     # address of string[high]
loop:  slt  $t0, $t1, $t3     # compare low and high addr
       beq  $t0 $zero, exit
       beq  $s3, $zero, exit  # exit if (matched == 0)
       lb   $t2, 0($t1)       # t2 = string[low]
       lb   $t4, 0($t3)       # t4 = string[high]
       beq  $t2, $t4, else
       addi $s3, $zero, 0     # matched = 0
       j endW                 # can be "j loop"
else:  addi $t1, $t1, 1       # low address increases
       addi $t3, $t3, -1      # high address decreases
endW:  j loop                 # end of while
exit:                         # outside of while
```

2. MIPS Bitwise Operations

Implement the following in MIPS assembly. Assume that integer variables **a**, **b** and **c** are mapped to registers $s0, $s1 and $s2 respectively. Each part is independent of all the other parts. **For bitwise instructions (e.g. ori, andi, etc),** any immediate values you use should be written in binary for this question. This is optional for non-bitwise instructions (e.g. addi, etc).

Note that bit 31 is the most significant bit (MSB) on the left, and bit 0 is the least significant bit (LSB) on the right, i.e.:

MSB					LSB
Bit 31	Bit 30	Bit 29	…	Bit 1	Bit 0

a. Set bits 2, 8, 9, 14 and 16 of **b** to 1. Leave all other bits unchanged.
To set bits, we create a "mask" with 1's in the bit positions we want to set. Since bit 16 is in the upper 16 bits of the register, we need to use lui to set it.

lui $t0, 1   # Sets bit 16 of $t0.
ori $t0, $t0, 0b0100001100000100 # Set bits 14, 9, 8
and 2. or $s1, $s1, $t0

b. Copy over bits 1, 3 and 7 of **b** into **a**, without changing any other bits of **a**. # We use the property that x AND 1 = x to copy out the values of
# bits 7, 3 and 1 of b into $t0. Note that we zero all other bits
# so that they don't change anything in $s0 when we OR
later on. andi $t0, $s1, 0b0000000010001010

# We use the property of x OR 0 = x to copy in
# the bits into a, so we prepare a by zero-ing bits 7, 3 and 1.
# To do this we need the mask 1111111111111111
1111111101110101 lui $t1, 0b1111111111111111
ori $t1, $t1, 0b1111111101110101
and $s0, $s0, $t1

# Now OR together a and $t0 to copy over
the bits or $s0, $s0, $t0

c. Make bits 2, 4 and 8 of **c** the inverse of bits 1, 3 and 7 of **b** (i.e. if bit 1 of **b** is 0, then bit 2 of **c** should be 1; if bit 1 of **b** is 1, then bit 2 of **c** should be 0), without changing any other bits of **c**. # We use the property that x XOR 1 = ~x to flip the values of bits 7, 3 and 1.
xori $t0, $s1, 0b10001010

# Zero every bit except 7, 3
and 1. andi $t0, $t0,
0b10001010

# Shift left one position: bit 1 becomes 0, bit 1 becomes bit 2, bit 3 becomes bit 4, and bit 7
become
s bit 8.
sll $t0,

$t0, 1

# Now, to clear bits 8, 4 and 2 of c,
# we need the mask 0b1111111111111111
1111111011101010 lui $t1, 0b1111111111111111
ori $t1, $t1, 0b1111111011101011
and $s2, $s2, $t1

# and we OR the new c
with $t0 or $s2, $s2, $t0

3. MIPS Arithmetic
Write the following in MIPS Assembly, using as few instructions as possible. You may rewrite the equations if necessary to minimize instructions.

In all parts you can assume that integer variables **a**, **b**, **c** and **d** are mapped to registers $s0, $s1, $s2 and $s3 respectively. Each part is independent of the others.

a. c = a + b

add $s2, $s0, $s1

b. d = a + b – c

```
add $s3, $s0, $s1     # d = a + b
sub $s3, $s3, $s2     # d = (a + b) - c
```

c. c = 2b + (a – 2)

add $s2, $s1, $s1 # c = 2b (alternatively, can do a shift
left 1 bit) addi $t0, $s0, -2 # $t0 = a - 2
add $s2, $s2, $t0 # c = 2b + (a – 2)

d. d = 6a + 3(b – 2c)

Note to TAs: Students may find better solutions than this. Check to ensure that they achieve the equation above.

Rewrite:
d = 6a +
3b – 6c
Factoriz
e out 3

d = 3(2a + b – 2c)
= 3(2a – 2c + b)
= 3(2(a – c) + b)

sub $t0, $s0, $s2 # t0 = a – c
sll $t0, $t0, 1 # t0 =
2(a – c) add $t0, $t0, $s1 #
t0 = 2(a – c) + b
sll $t1, $t0, 2 # t1 = 4(2(a
– c) + b) sub $s3, $t1, $t0 # d =
3(2(a – c) + b)

4. [AY2013/14 Semester 2 Exam]
The mysterious MIPS code below assumes that **$s0 is a 31-bit binary sequence**, i.e. the MSB (most significant bit) of **$s0** is assumed to be zero at the start of the code.

```
      add   $t0, $s0, $zero    # make a copy of $s0 in $t0
      lui   $t1, 0x8000
lp:   beq   $t0, $zero, e
      andi  $t2, $t0, 1
      beq   $t2, $zero, s
      xor   $s0, $s0, $t1
s:    srl   $t0, $t0, 1
      j     lp
e:
```

a) For each of the following initial values in register **$s0** at the beginning of the code, give the hexadecimal value of the content in register $**s0** at the end of the code.

i. Decimal value **31**.
ii. Hexadecimal value **0x0AAAAAAA**.

b) Explain the purpose of the code in one sentence.

Answers:
a. i. $s0 = 0x8000 001F
ii. **$s0 = 0x0AAA AAAA**

b. The code sets bit 31 of $s0 to 1 if there are odd number of '1' in $s0 initially, or 0 if there are even number of '1'. (This is called the even parity bit.)

IT5002 Computer Systems and Applications
Tutorial 3

Tutorial Questions

Questions 1 and 2 refer to the complete datapath and control design in lectures 7 and 8. For your convenience, the complete datapath is attached at the end of this tutorial.

1. Let us perform a complete trace to understand the working of the complete datapath and control implementation. Given the following three hexadecimal representations of MIPS instructions:

   i. `0x8df80000: lw $24, 0($15)`
   ii. `0x1023000C: beq $1, $3, 12`
   iii. `0x0285c822: sub $25, $20, $5`

   For each instruction encoding, do the following:

   a. Fill in the tables below. The first table concerns with the various data (information) at each of the datapath elements, while the second table records the control signals generated. Use the notation $8 to represent register number 8, [$8] to represent the content of register number 8 and Mem(X) to represent the memory data at address X.

Registers File				ALU		Data Memory	
RR1	RR2	WR	WD	Opr1	Opr2	Address	Write Data

[Wr = Write; Rd = Read; M = Mem; R = Reg]

RegDst	RegWr	ALUSrc	MRd	MWr	MToR	Brch	ALUop	ALUctrl

   b. Indicate the value of the PC after the instruction is executed.

Answers:

Only values in RED and **BOLD** font are actually utilized in the execution.

i. `0x8df80000 = lw $24, 0($15);` next PC = PC+4

Registers File	ALU	Data Memory

RR1	RR2	WR	WD	Opr1	Opr2	Address	Write Data
$15	$24	**$24**	**MEM([$15]+0)**	**[$15]**	**0**	**[$15]+0**	[$24]

RegDst	**RegWr**	**ALUSrc**	**MRd**	**MWr**	**MToR**	**Brch**	**ALUop**	**ALUctrl**
0	1	1	1	0	1	0	00	0010

`ii. 0x1023000C = beq $1, $3, 12;` next PC = PC+4 or (PC+4)+(12×4)

Registers File				**ALU**		**Data Memory**	
RR1	RR2	WR	WD	Opr1	Opr2	Address	Write Data
$1	**$3**	$3 or $0	[$1]-[$3] *or* random value	**[$1]**	**[$3]**	[$1]-[$3]	[$3]

RegDst	**RegWr**	**ALUSrc**	**MRd**	**MWr**	**MToR**	**Brch**	**ALUop**	**ALUctrl**
X	0	0	0	0	X	1	01	0110

`iii. 0x0285c822 = sub $25, $20, $5;` next PC = PC+4

Registers File				**ALU**		**Data Memory**	
RR1	RR2	WR	WD	Opr1	Opr2	Address	Write Data
$20	**$5**	**$25**	**[$20]-[$5]**	**[$20]**	**[$5]**	[$20]-[$5]	[$5]

RegDst	**RegWr**	**ALUSrc**	**MRd**	**MWr**	**MToR**	**Brch**	**ALUop**	**ALUctrl**
1	1	0	0	0	0	0	10	0110

2. With the complete datapath and control design, it is now possible to estimate the latency (time needed for a task) for the various type of instructions. Given below are the resource latencies of the various hardware components (ps = picoseconds = 10^{-12} second):

Inst-Mem	**Adder**	**MUX**	**ALU**	**Reg-File**	**Data-Mem**	**Control/ ALUControl**	**Left-shift/ Sign-Extend/ AND**
400ps	100ps	30ps	120ps	200ps	350ps	100ps	20ps

Give the estimated latencies for the following MIPS instructions:

(a) "SUB" instruction (e.g. `sub $25, $20, $5`)
(b) "LW" instruction (e.g. `lw $24, 0($15)`)
(c) "BEQ" instruction (e.g. `beq $1, $3, 12`)

What do you think the **cycle time** should be for this particular processor implementation?

Hint: First, you need to find out the **critical path** of an instruction, i.e. the path that takes the longest time to complete. Note that there could be several parallel paths that work more or less simultaneously.

Answers:

[To Tutor] It is easier to note the timing on the datapath & control diagram and show them the critical path. Strongly suggest to use the projector to show the full diagram.

(a) SUB instruction (R-type):

Critical Path:
I-Mem □Reg.File □ MUX(ALUSrc) □ ALU □ MUX(MemToReg) □ Reg.File

Note: I-MEM □ Control is a parallel path, the earliest signal needed is the ALUSrc. So, as long as the Control latency is lesser than Reg.File access latency, then it will not be in the critical path. Once the signal is generated, the Control latency will no longer affect the overall delays.

Similarly, there is another path to calculate the next PC (I-MEM □ Control □ AND □ MUX(PCSrc) which is again not critical to the overall latency.

Latency = 400 + 200 + 30 + 120 + 30 + 200 = 980ps

(b) LW instruction:

Critical Path:
I-Mem □ Reg.File □ ALU □ DataMem □ MUX(MemToReg) □ Reg.File

Latency = 400 + 200 + 120 + 350 + 30 + 200 = 1300ps

Note: The path I-Mem □ Immediate □ MUX(ALUSrc) occurs simultaneously with the above.

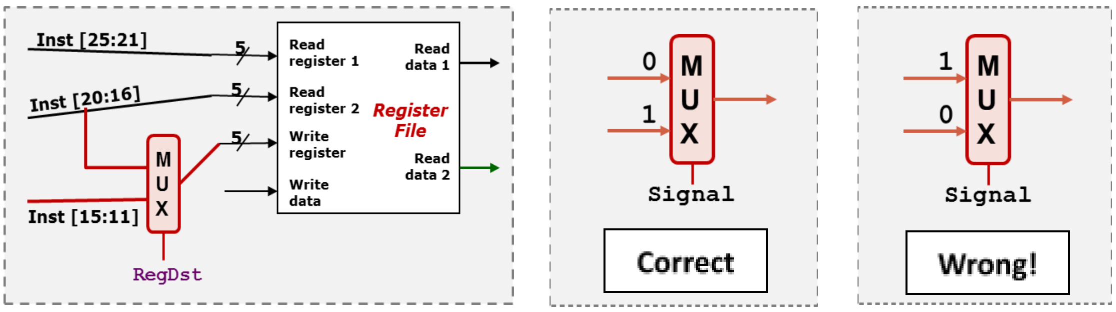
(c) BEQ instruction:

Critical Path:
I-Mem □ Reg.File □ MUX(ALUSrc) □ ALU □ AND □ MUX(PCSrc)

Latency = 400 + 200 + 30 + 120 + 20 + 30 = 800ps

Since LW has the longest latency. The overall cycle time of the whole machine is determined by LW, i.e. at least 1300ps.

3. [AY2013/14 Semester 2 Term Test #2]
Mr. De Blunder made a ***huge*** mistake while making his own non-pipelined MIPS processor. He accidentally **swapped the two input ports for the RegDst multiplexer:**



For each of the following instructions, give:

i. One example where the incorrect processor still gives the **right execution result.**
ii. One example where the incorrect processor gives the **wrong execution result**.

If there is no suitable answer, please indicate "No Answer".

(a) `add` (Addition)
(b) `lw` (Load Word)
(c) `beq` (branch-if-equal), provide the branch offset as immediate value.

Answers:

Many possible answers, so only a few are given here.

(a)

i. **`add X, Y, X`** (i.e. RT and RD are the same.)

ii. **`add X, Y, Z`**

(b)

i. **`lw $RT, {"$RT", followed by 11 bits}($1)`**

– the MSB 5 bits of immediate == RT

A few examples (assuming that the 11 bits are 0s):

RT	RT	RT	RT	RT	0	0	0	0	0	0	0	0	0	0	0

$a0 □ Immediate = 0x2000 = 8192 (i.e. lw $a0, 8192($any))

0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

$t0 □ Immediate = 0x4000 = 16384

0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

$s0 □ Immediate = 0x8000 = -(0x8000) = -32768

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

$t8 □ Immediate = 0xD000 = -(0x3000) = -12288

1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0

ii. Anything other than the above.

(c)

i. Any instructions (as the error would have no impact on branch instructions).

ii. No answer.

4. Suppose the four stages in some 4-stage pipeline take the following timing: 2ns, 3ns, 4ns, and 2ns. Given 1000 instructions, what is the speedup (in two decimal places) of the pipelined processor compared to the non-pipelined single-cycle processor?

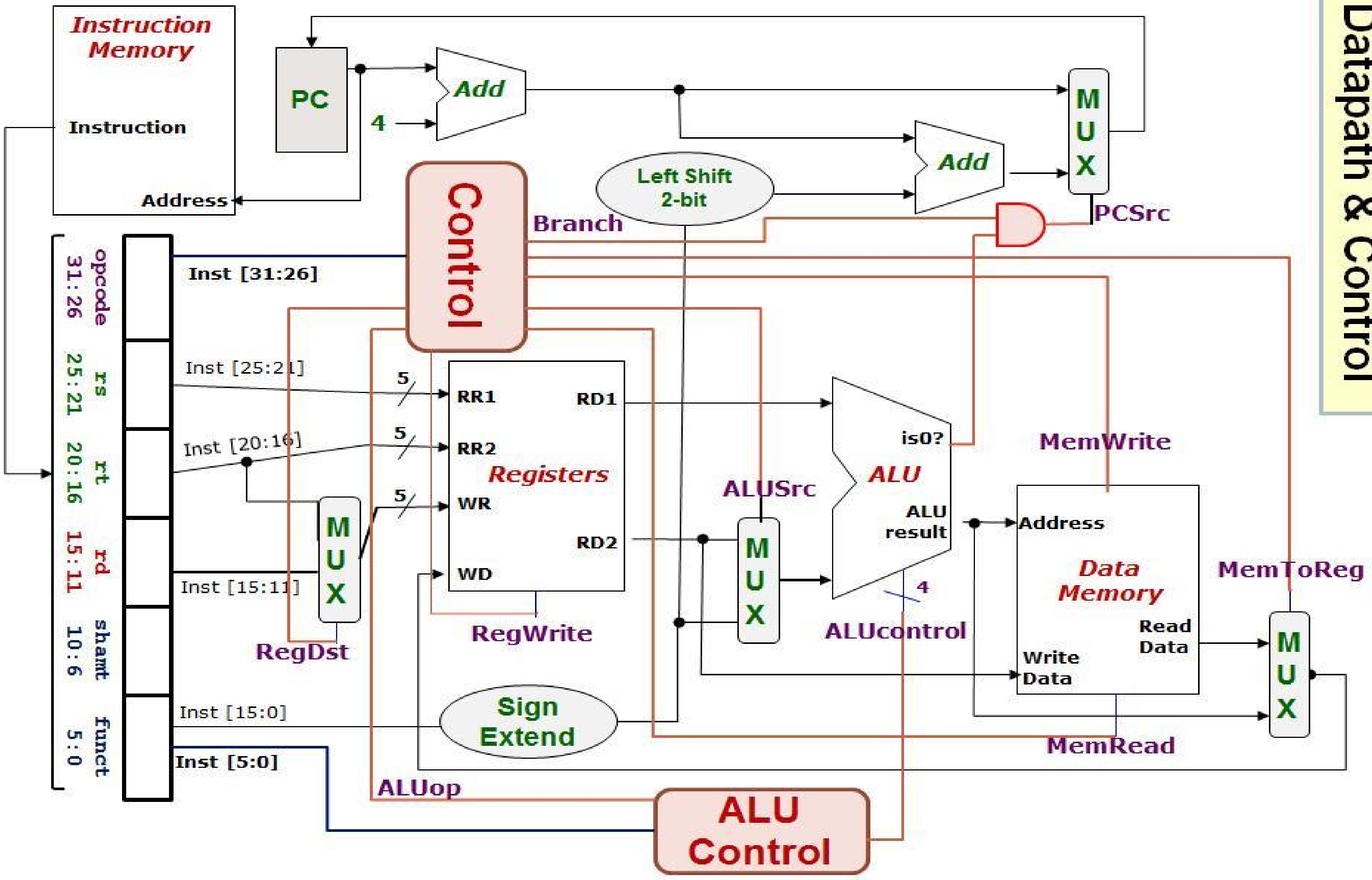
Answer:

Non-pipelined: 11ns x 1000 = 11,000ns.

In a pipelined system, each stage takes 4ns.

Pipelined: 12ns + (1,000 x 4) ns = 12 ns + 4,000ns = 4,012ns. Speedup = 11,000/4,012 = **2.74**

Datapath & Control



IT5002 Computer Systems and Applications
Tutorial 4

1. Suppose the five stages in some 5-stage pipeline (fetch-decode-ALU-memory-writeback) take the following timing: 2ns, 3ns, 4ns, 8ns and 2ns. Given 1000 instructions, what is the speedup of running on a pipelined processor versus:

In a pipelined CPU, the number of cycles taken can be calculated from:

1. It takes 4 cycles for the first instruction to complete, since it is a 4-stage pipeline and each stage takes one cycle.

2. It takes 1 cycle each for the remaining N-1 instructions.

3. Total cycles = 4 + N – 1 = 4 + 999 = 1003 cycles.

4. The cycle time must be the slowest of all the stages (otherwise the slowest stage will not have time to complete) = 8ns

5. Total time = 8024 ns

   i. A single-cycle non-pipelined CPU.

Cycle time must be equal to the sum of the time of all the stages = 2 + 3 + 4 + 8 + 2 = 19ns.
Total time = 1000 x 19 = 19000 ns.
Speedup = 19000 / 8024 = 2.37 times.

ii. A multi-cycle non-pipelined CPU, given that 70% of the instructions are arithmetic, 25% are branch instructions, and 2% are loads and 3% are stores.

We need to first work out the instruction times for each type of instruction based on the stages they use. Each stage must take 8ns, the timing of the slowest stage.

Instruction	IF 2ns	ID 3ns	ALU 4ns	MEM 8ns	WB 2ns	Time ns
Arithmetic	X	X	X		X	$4 \times 8 = 32$
Branch	X	X	X			$3 \times 8 = 24$
Load	X	X	X	X	X	$5 \times 8 = 40$
Store	X	X	X	X		$4 \times 8 = 32$

Total cycle time = $700 \times 32 + 250 \times 24 + 20 \times 40 + 30 \times 32$
= 22,400 + 6000 + 800 + 960
= 30,160 ns

Speedup = 30,160 / 8024 = 3.76 times.

2. Let's try to understand pipeline processor by doing a detailed trace. Suppose the pipeline registers (also known as pipeline latches) store the following information:

IF / ID	
No Control Signal	--------
PC+4	
OpCode	
Rs	
Rt	
Rd	
Funct	
Imm(16)	

ID / EX	
MToR	
RegWr	
MemRd	
MemWr	
Branch	
RegDst	
ALUsrc	
ALUop	
PC+4	
ALUOpr1	
ALUOpr2	
Rt	
Rd	
Imm(32)	

EX / MEM	
MToR	
RegWr	
MemRd	
MemWr	
Branch	
BrcTgt	
isZero?	
ALURes	
ALUOpr2	
DstRNum	

MEM / WB	
MToR	
RegWr	
MemRes	
ALURes	
DstRNum	

Show the progress of the following instructions through the pipeline stages by filling in the content of pipeline registers.

i. `0x8df80000  # lw $24, 0($15)      #Inst.Addr = 0x100`
ii. `0x1023000C  # beq $1, $3, 12      #Inst.Addr = 0x100`
iii. `0x0285c822  # sub $25, $20, $5    #Inst.Addr = 0x100`

Assume that registers 1 to 31 have been initialized to a value that is equal to 101 + its register number. i.e. [$1] = 102, [$31] = 132 etc. You can put "X" in fields that are irrelevant for that instruction. Do note that in reality, these fields are actually generated but not utilized.
Part (i) has been worked out for you.

i. `0x8df80000  # lw $24, 0($15)      #Inst.Addr = 0x100`

IF / ID	
No Control Signal	--------
PC+4	0x104
OpCode	0x23
Rs	$15
Rt	$24
Rd	X
Funct	X
Imm(16)	0

ID / EX	
MToR	1
RegWr	1
MemRd	1
MemWr	0
Branch	0
RegDst	0
ALUsrc	1
ALUop	00
PC+4	0x104
ALUOpr1	116
ALUOpr2	X
Rt	$24
Rd	X
Imm(32)	0

EX / MEM	
MToR	1
RegWr	1
MemRd	1
MemWr	0
Branch	0
BrcTgt	X
isZero?	X
ALURes	116
ALUOpr2	X
DstRNum	$24

MEM / WB	
MToR	1
RegWr	1
MemRes	Mem(116)
ALURes	X
DstRNum	$24

Answers:

ii. `0x1023000C # beq $1, $3, 12 #Inst.Addr = 0x100`

IF / ID	
No Control Signal	-------
PC+4	0x104
OpCode	4
Rs	$1
Rt	$3
Rd	X
Funct	X
Imm(16)	12

ID / EX	
MToR	X
RegWr	0
MemRd	0
MemWr	0
Branch	1
RegDst	X
ALUsrc	0
ALUop	01
PC+4	0x104
ALUOpr1	102
ALUOpr2	104
Rt	X
Rd	X
Imm(32)	12

EX / MEM	
MToR	X
RegWr	0
MemRd	0
MemWr	0
Branch	1
BrcTgt	0x134
isZero?	0
ALURes	-2
ALUOpr2	X
DstRNum	X

MEM / WB	
MToR	X
RegWr	0
MemRes	X
ALURes	X
DstRNum	X

iii. `0x0285c822 # sub $25, $20, $5 #Inst.Addr = 0x100`

IF / ID	
No Control Signal	-------
PC+4	0x104
OpCode	0
Rs	$20
Rt	$5
Rd	$25
Funct	22
Imm(16)	X

ID / EX	
MToR	0
RegWr	1
MemRd	0
MemWr	0
Branch	0
RegDst	1
ALUsrc	0
ALUop	10
PC+4	0x104
ALUOpr1	121
ALUOpr2	106
Rt	X
Rd	$25
Imm(32)	X

EX / MEM	
MToR	0
RegWr	1
MemRd	0
MemWr	0
Branch	0
BrcTgt	X
isZero?	X
ALURes	15
ALUOpr2	X
DstRNum	$25

MEM / WB	
MToR	0
RegWr	1
MemRes	X
ALURes	15
DstRNum	$25

3. Given the following three formulas:

$$CT_{seq} = \sum_{k=1}^{N} T_k$$

$$CT_{pipeline} = \max(T_k) + T_d$$

$$Speedup_{pipeline} = \frac{CT_{seq} \times InstNum}{CT_{pipeline} \times (N + InstNum - 1)}$$

For each of the following processor parameters, calculate CT_{seq}, $CT_{pipeline}$ and $Speedup_{pipeline}$ (to two decimal places) for 10 instructions and for 10 million instructions.

	Stages Timing (for 5 stages, in ps)	Latency of pipeline register (in ps)
a.	300, 100, 200, 300, 100 (slow memory)	0
b.	200, 200, 200, 200, 200	40
c.	200, 200, 200, 200, 200 (ideal)	0

Answers:

	CT_{seq}	$CT_{pipeline}$	Speedup (10 inst)	Speedup (10m inst)
a.	**1000ps**	**300ps**	(1000×10)/(300×14) = **2.38**	(1000×10m)/(300 (10m+4)) = **3.33**
b.	**1000ps**	**240ps**	(1000×10)/(240×14) = **2.98**	(1000×10m)/(240×(10m+4)) = **4.17**
c.	**1000ps**	**200ps**	(1000×10)/(200×14) = **3.57**	(1000×10m)/(200×(10m+4)) = **5.00**

4. Here is a series of address references in decimal: 4, 16, 32, 20, 80, 68, 76, 224, 36, 44, 16, 172, 20, 24, 36, and 68 in a MIPS machine. Assuming a **direct-mapped cache** with 16 one-word blocks that is initially empty, label each address reference as a hit or miss and show the content of the cache.

   You may write the data word starting at memory address X as M[X]. (For example, dataword starting at memory address 12 is written as M[12]. This implies that the word includes the 4 bytes of data at addresses 12, 13, 14 and 15.) You may write the tag values as decimal numbers. If a block is replaced in the cache, cross out the corresponding content in the cache, and write the new content over it.

Answer:

Since this is a MIPS machine, a word consists of 4 bytes. Should first work out the tag, index, and offset fields:

26 bits	4 bits	2
Tag	Index	Offse

4:	00…00	0001	00 ← Miss
16:	00…00	0100	00 ← Miss
32:	00…00	1000	00 ← Miss
20:	00…00	0101	00 ← Miss
80:	00…01	0100	00 ← Miss
68:	00…01	0001	00 ← Miss
76:	00…01	0011	00 ← Miss
224:	00…11	1000	00 ← Miss
36:	00…00	1001	00 ← Miss
44:	00…00	1011	00 ← Miss
16:	00…00	0100	00 ← Miss
172:	00…10	1011	00 ← Miss
20:	00…00	0101	00 ← Hit
24:	00…00	0110	00 ← Miss
36:	00…00	1001	00 ← Hit
68:	00…01	0001	00 ← Hit

Cache block	Valid bit	Tag	Word
0	0		
1	~~0~~ 1	~~0~~ 1	~~M[4]~~ M[68]
2	0		
3	~~0~~ 1	1	M[76]
4	~~0~~ 1	~~0 1~~ 0	~~M[16] M[80]~~ M[16]
5	~~0~~ 1	0	M[20]
6	~~0~~ 1	0	M[24]
7	0		
8	~~0~~ 1	~~0~~ 3	~~M[32]~~ M[224]
9	~~0~~ 1	0	M[36]
10	0		
11	~~0~~ 1	~~0~~ 2	~~M[44]~~ M[172]
12	0		
13	0		
14	0		
15	0		

IT5002 Computer Systems and Applications
Tutorial 5
Cache
SUGGESTED SOLUTIONS

Here is a series of address references in decimal: 4, 16, 32, 20, 80, 68, 76, 224, 36, 44, 16, 172, 20, 24, 36, and 68 in a MIPS machine. Assuming a **direct-mapped cache** with 16 one-word blocks that is initially empty, label each address reference as a hit or miss and show the content of the cache.

You may write the data word starting at memory address X as M[X]. (For example, data word starting at memory address 12 is written as M[12]. This implies that the word includes the 4 bytes of data at addresses 12, 13, 14 and 15.) You may write the tag values as decimal numbers. If a block is replaced in the cache, cross out the corresponding content in the cache, and write the new content over it.

Answer:

Since this is a MIPS machine, a word consists of 4 bytes.

Should first work out the tag, index, and offset fields:

26 bits	4 bits	2
Tag	Index	Offset

4:	00…00	0001	00	← Miss
16:	00…00	0100	00	← Miss
32:	00…00	1000	00	← Miss
20:	00…00	0101	00	← Miss
80:	00…01	0100	00	← Miss
68:	00…01	0001	00	← Miss
76:	00…01	0011	00	← Miss
224:	00…11	1000	00	← Miss
36:	00…00	1001	00	← Miss
44:	00…00	1011	00	← Miss
16:	00…00	0100	00	← Miss
172:	00…10	1011	00	← Miss
20:	00…00	0101	00	← Hit
24:	00…00	0110	00	← Miss
36:	00…00	1001	00	← Hit
68:	00…01	0001	00	← Hit

Cache block	Valid bit	Tag	Word
0	0		
1	~~0~~ 1	~~0~~ 1	~~M[4]~~ M[68]
2	0		
3	~~0~~ 1	1	M[76]
4	~~0~~ 1	~~0~~ ~~1~~ 0	~~M[16]~~ ~~M[80]~~ M[16]
5	~~0~~ 1	0	M[20]
6	~~0~~ 1	0	M[24]
7	0		
8	~~0~~ 1	~~0~~ 3	~~M[32]~~ M[224]
9	~~0~~ 1	0	M[36]
10	0		
11	~~0~~ 1	~~0~~ 2	~~M[44]~~ M[172]
12	0		
13	0		
14	0		
15	0		

2. Use the series of references given in question 1 above: 4, 16, 32, 20, 80, 68, 76, 224, 36, 44, 16, 172, 20, 24, 36, and 68 in a MIPS machine. Assuming a **two-way set-associative cache** with two-word blocks and a total size of 16 words that is initially empty, label each address reference as a hit or miss and show the content of the cache. Assume **LRU** replacement policy.

   You may write the data word starting at memory address X as M[X]. (For example, data word starting at memory address 12 is written as M[12]. This implies that the word includes the 4 bytes of data at addresses 12, 13, 14 and 15.) You may write the tag values as decimal numbers. If a block is replaced in the cache, cross out the corresponding content in the cache, and write the new content over it.

   Answer:

   Since this is a MIPS machine, a word consists of 4 bytes or 32 bits.

   Should first work out the tag, set index, and offset fields:

27 bits	2 bits	3
Tag	Set Index	Offset

4:	00…000	00	100	← Miss
16:	00…000	10	000	← Miss
32:	00…001	00	000	← Miss
20:	00…000	10	100	← Hit
80:	00…010	10	000	← Miss
68:	00…010	00	100	← Miss
76:	00…010	01	100	← Miss
224:	00…111	00	000	← Miss
36:	00…001	00	100	← Miss
44:	00…001	01	100	← Miss
16:	00…000	10	000	← Hit
172:	00…101	01	100	← Miss
20:	00…000	10	100	← Hit
24:	00…000	11	000	← Miss
36:	00…001	00	100	← Hit
68:	00…010	00	100	← Miss

Cache set	Valid bit	Tag	Word0	Word1	Valid bit	Tag	Word0	Word1
0	~~0~~ 1	~~0~~ ~~2~~ 1	~~M[0]~~ ~~M[64]~~ M[32]	~~M[4]~~ ~~M[68]~~ M[36]	~~0~~ 1	~~1~~ ~~7~~ 2	~~M[32]~~ ~~M[224]~~ M[64]	~~M[36]~~ ~~M[228]~~ M[68]
1	~~0~~ 1	~~2~~ 5	~~M[72]~~ M[168]	~~M[76]~~ M[172]	~~0~~ 1	1	M[40]	M[44]
2	~~0~~ 1	0	M[16]	M[20]	~~0~~ 1	2	M[80]	M[84]
3	~~0~~ 1	0	M[24]	M[28]	0			

3. Although we use only data memory as example in the cache lecture, the principle covered is equally applicable to the instruction memory. This question takes a look at both the instruction cache and data cache.

   The code below is a ***palindrome checker*** with the following variable mappings:

   low → \$s0, high→ \$s1, matched → \$s3, base of string[]→ \$s4, size → \$s5

#	Code	Comment
i0	`[some instruction]`	
i1	`addi $s0, $zero, 0`	`# low = 0`
i2	`addi $s1, $s5, -1`	`# high = size-1`
i3	`addi $s3, $zero, 1`	`# matched = 1`
	`loop:`	
i4	`slt  $t0, $s0, $s1`	`# (low < high)?`
i5	`beq  $t0, $zero, exit`	`# exit if (low >= high)`
i6	`beq  $s3, $zero, exit`	`# exit if (matched == 0)`
i7	`add  $t1, $s4, $s0`	`# address of string[low]`
i8	`lb   $t2, 0($t1)`	`# t2 = string[low]`
i9	`addi $t3, $s4, $s1`	`# address of string[high]`
i10	`lb   $t4, 0($t3)`	`# t4 = string[high]`
i11	`beq  $t2, $t4, else`	
i12	`addi $s3, $zero, 0`	`# matched = 0`
i13	`j    endW`	`# can be "j loop"`
	`else:`	
i14	`addi $s0, $s0, 1`	`# low++`
i15	`addi $s1, $s1, -1`	`# high–`
	`endW:`	
i16	`j    loop`	`# end of while`
	`exit:`	
i17	`[some instruction]`	

Parts (a) to (d) assume that instruction i0 is stored at memory address 0x0.

(a) Instruction cache: **Direct mapped with 2 blocks of 16 bytes each** (i.e. each block can hold 4 consecutive instructions).

Starting with an empty cache, the fetching of instruction i1 will cause a cache miss. After the cache miss is resolved, we now have the following instructions in the instruction cache:

Instruction Cache Block 0	[i0, **i1**, **i2**, **i3**]
Instruction Cache Block 1	[empty]

Fetching of i2 and i3 are all cache hits as they can be found in the cache.

Assuming the string being checked is a palindrome. Show the instruction cache block content **at the end of the 1st iteration (i.e. up to instruction i16).**

Answer:

Instruction Cache Block 0	**[i16, ……..]**
Instruction Cache Block 1	**[i12, i13, i14, i15]**

Working: Instructions executed = i1 to i11, i14 to i16

Block #0, Cache index = 0	[i0, i1, i2, i3]
Block #1, Cache index = 1	[i4, i5, i6, i7]
Block #2, Cache index = 0	[i8, i9, i10, i11]
Block #3, Cache index = 1	[i12, i13, i14, i15]
Block #4, Cache index = 0	[i16, other….]

(b) If the loop is executed for a total of 10 iterations, what is the total number of cache hits (i.e. after the 10th "j loop" is fetched)?

Answer:

Working (1st Iteration):

i1	i2	i3	i4	i5	i6	i7	i8	i9	i10	i11	i14	i15	i16
M	H	H	M	H	H	H	M	H	H	H	M	H	M

Working (2nd iteration onward):

i4	i5	i6	i7	i8	i9	i10	i11	i14	i15	i16
M	H	H	H	M	H	H	H	M	H	M

Total hits = 9 (1st iteration) + 7×9 (remaining 9 iterations) = **72**

(c) Suppose we change the instruction cache to:

- **Direct mapped with 4 blocks of 8 bytes each** (i.e. each block can hold 2 consecutive instructions).

Assuming the string being checked is a palindrome. Show the instruction cache block content **at the end of the 1st iteration (i.e. up to instruction i16).**

Answer:

Instruction Cache Block 0	**[i16, …]**
Instruction Cache Block 1	**[i10, i11]**
Instruction Cache Block 2	**[i4, i5]**
Instruction Cache Block 3	**[i14, i15]**

Working:

First, find out the block information for the full code:

Block #0, Cache index = 0	[i0, i1]
Block #1, Cache index = 1	[i2, i3]
Block #2, Cache index = 2	[i4, i5]
Block #3, Cache index = 3	[i6, i7]
Block #4, Cache index = 0	[i8, i9]
Block #5, Cache index = 1	[i10, i11]
Block #6, Cache index = 2	[i12, i13]
Block #7, Cache index = 3	[i14, i15]
Block #8, Cache index = 0	[i16, …]

Second, use the execution pattern to find out what is accessed, since we execute i1 to i11 (Block #0 to Block #5) then i14 to i16 (Block #7 and Block #8), we get the final cache content as shown. You should note that Block #6 [i12, i13] is not accessed in this particular execution.

(d) If the loop is executed for a total of 10 iterations, what is the total number of cache hits (i.e. after the 10th "j loop" is fetched)?

Answer:

Working (1st Iteration):

i1	i2	i3	i4	i5	i6	i7	i8	i9	i10	i11	i14	i15	i16
M	M	H	M	H	M	H	M	H	M	H	M	H	M

Working (2nd iteration onward):

i4	i5	i6	i7	i8	i9	i10	i11	i14	i15	i16
H	H	M	H	M	H	H	H	M	H	M

Total hits = 6 (1st iteration) + 7×9 (remaining 9 iterations) = **69**

Let us now turn to the study of **data cache**. We will assume the following scenario for parts (e) to (g):

- The string being checked is **64-character long**. The first character is located at location **0x1000**.
- The string is a palindrome (i.e. it will go through 32 iterations of the code).

(e) Given a **direct mapped data cache with 2 cache blocks, each block is 8 bytes**, what is the final content of the data cache at the end of the code execution (after the code failed the beq at i5)? Use **s[X..Y]** to indicate the data **string[X]** to **string[Y].**

Answer:

Data Cache Block #0	**s[32..39]**
Data Cache Block #1	**s[24..31]**

Access patterns = s[0], s[63], s[1], s[62], …, s[31], s[32]

Blocks information (blocks that can go into the same cache location are listed together):

Cache index = 0	s[0..7] [16..23] [32..39] [48..55]
Cache index = 1	s[8..15] [24..31] [40..47] [56..63]

(f) What is the hit rate of (e)? Give your answer in a fraction or a percentage correct to two decimal places.

Answer:

Observation: the access pattern nicely alternates between Block0-Block1 and Block1-Block0. So, in general, other than the first miss to bring in a block, the remaining 7 accesses on the block are all hits.

Hence, hit rate = **7/8** or **87.50%**

(g) Suppose the string is now **72-character long**, the first character is still located at location **0x1000** and the string is still a palindrome, what is the hit rate at the end of the execution?

Answer:

Access patterns = s[0], s[71], s[1], s[70], …, s[35], s[36]

Blocks information (blocks that can go into the same cache location are listed together):

Cache index = 0	s[0..7] [16..23] [32..39] [48..55] [64..71]
Cache index = 1	s[8..15] [24..31] [40..47] [56...63]

Observation: the access pattern is either Block0-Block0 or Block1-Block1. So, every access is a miss, except the last block [32..39]! This is an example of ***cache thrashing*** (you can imagine the cache is "beaten up" pretty badly ☺).

Hence, hit rate = **7/72** (the last 7 accesses on block [32..39]) or **9.72%**

IT5002 Computer Systems and Applications
Tutorial 6
SOLUTIONS

1. Using Google or otherwise, research and show how bootstrapping works on Windows, LINUX or MacOS (pick ONE OS to talk about).

   Here we will cover MacOS (Specifically OS X). This is a quick summary, for the full list please see http://osxdaily.com/2007/01/22/what-happens-in-the-mac-os-x-boot-process/

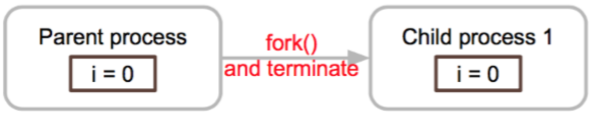
   - On power-on, the Mac starts executing code in the Unified Extensible Firmware Interface (EFI), which is in ROM. EFI is not unique to Macs and it replaces BIOS (Basic Input Output System) that was used in earlier computers.
   - EFI initializes all the hardware, and loads and hands control over to /System/Library/CoreServices/BootX, which is the OS X boot loader. BootX and Mach (see later) all run at a high kernel security level (kernel mode).
   - BootX loads the kernel, and starts loading up all the device drivers that are needed to run the screen, touchpad, WiFi interface, etc on your Mac.
   - The kernel init routine is started, and at this point the Mac's firmware hands control over to OS X. Various data structures are initialized.
   - I/O Kit is started, giving the OS access to devices on the Mac via their drivers.
   - The Mach service, an extensible microkernel communications service, is started. Mach provides various services like communication ports, remote procedure call (RPC) support, notifications, interprocess communications, etc.
   - The kernel switches to a standard user security level (user mode), and starts /sbin/init, which becomes the first process in the system (init is the master process of any UNIX system)
   - Init looks at rc.boot and rc.common to start up user-level scripts. These scripts do things like start up servers, or initialize software systems the user needs.
   - Init runs fsck, which determines if any full filesystem check is needed.
   - The main filesystem /dev/fd is mounted and is now accessible.
   - /etc/rc.cleanup is executed to do any filesystem cleaning, if needed.
   - Various network services are started.
   - The swap partition is mounted and the virtual memory system is started.
   - The OS X graphical shell is started.

2. How many processes are created in the following program? Include the original process created when the program is first run.

```
for(int i=0; i<10; i++) fork();
```

It is extremely tedious to work this out by hand. Fortunately there is an easier way. We start first with for(i=0; i<1; i++):

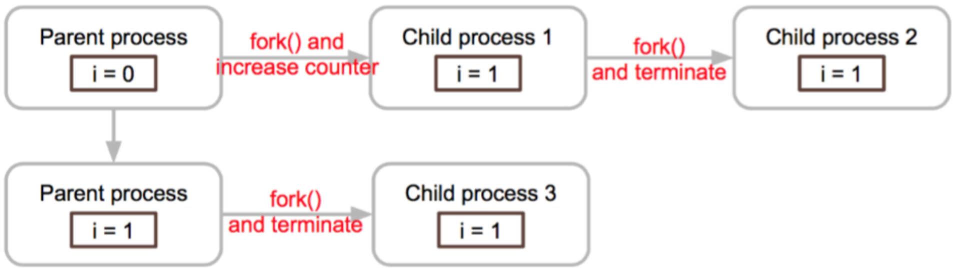
Analysis for (int i = 0; i < 1; i++):



Total number of processes: 2

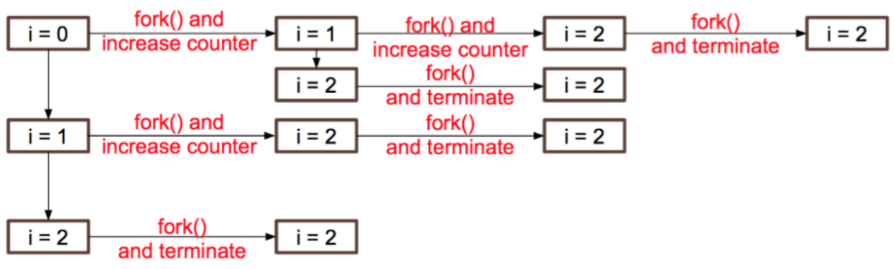
We do the same for for(i=0; i<2; i++):

Analysis for (int i = 0; i < 2; i++):



Total number of processes: 4

Analysis for (int i = 0; i < 3; i++):



Total number of processes: 8

```
We can see that the forks form essentially a binary
tree, and that for the general case n, we will have
2^n forks.
```

3. Explain, with an example, why the highest priority task can still be interrupted by the lowest priority interrupt, and how this affects ISR design. (Hint: Interrupts are implemented in hardware in the CPU itself).

This is because task priorities are seen only by the OS. As far as the CPU is concern, it is “just another program” that is running, with instructions being fetched and executed. On the other hand interrupt lines are checked at the end of each instruction execution cycle, so interrupts are always serviced regardless of their priority level (and of the priority level of the running task).

4. We know that to support multitasking we need to save the registers, program counter, stack pointers and machine status word (SREG in the lecture notes). What other pieces of information does the OS need to save about a process? Explain each piece.

   File handles / Open File Table: These are data structures that maintain information about files that are opened by the process, like the location that a process is inside the file, access rights to the file, file open modes, etc.
   Pending signals: A “signal” is an OS message to the process. For example SIGINT interrupts a process (pressing CTRL-C triggers a SIGINT), SIGTERM terminates a process (triggered using kill process_id), SIGKILL kills a process (triggered using kill -9 process_id), SIGSUSPEND suspends a process (triggered when you press CTRL-Z) etc. These signals either trigger default behaviors (e.g. breaking a process), or can be caught and handled by signal handlers in the program. More information: The difference between SIGTERM and SIGKILL is that SIGTERM allows a process to gracefully shut down its child processes, while SIGKILL kills the process immediately and can leave orphan child processes. Be kind. Always use SIGTERM and not SIGKILL. Don’t leave orphans behind!
   Process Running State: Whether the process is suspending, ready, running, terminated, etc.
   Accounting Information: How much CPU time the process has used, how much disk space, network activity, etc.
   Process ID: Unique number identifying the process. Etc.

5. Given four batch jobs T1, T2, T3 and T4 with running times of 190 cycles, 300 cycles, 30 cycles and 130 cycles, find:

   i) The average waiting time to run if the jobs are executed in order.

Task	Waiting time
T1	0
T2	0+190 = 190
T3	0 + 190 + 300 = 490
T4	0 + 190 + 300 + 30 = 520

Total = 0 + 190 + 490 + 520 = 1200 cycles
Average = 300 cycles waiting time.

   ii) The average time to run if the jobs are executed SJF.

Task	Waiting time
T3	0
T4	0 + 30 = 30
T1	0 + 30 + 130 = 160
T2	0 + 30 + 130 + 190 = 350

Total waiting time = 0 + 30 + 160 + 350 = 540 cycles
Average = 135 cycles

From your answer and otherwise, explain the advantage and disadvantage of SJF. In particular, what if the number of jobs running is not fixed and new jobs can be added at any time?

SJF means that jobs on average wait less time to be executed, which means that if you submitted many jobs, you will get, on average, a faster response. This makes the system "feel better" because you don't have to wait too long to get a result.

SJF is prone to starvation if there's a long task, and someone keeps submitting short tasks.

IT5002 Tutorial 7

Race Conditions

Question 1

We have the following two threads:

```
int x = 5; // Shared variable

void *thread1(void *p)
{
        while(1)
        {
                x++;
                .. Other lines of code ..
        }
}

void *thread2(void *p)
{
        while(1)
        {
                x*=2;
                .. Other lines of code ..
        }
}
```

a. Write down the x++ and x*=2 in assembly on a CPU with load-store architecture (i.e. ALU only operates on registers. You must first load variables in memory into registers. You must then write the registers back to the variables in memory) You can choose any kind of assembly language, even one that doesn't exist, as long as it is a load-store architecture.

   Thread 1:

```
loop:
        lw $r0, x
        addi $r0, $r0, 1
        sw $r0, x
        jmp loop
```

   Thread 2:

```
loop:
        lw $r0, x
        shl $r0, $r0, 1
        sw $r0, x
        jmp loop
```

b. Based on your answer to a., what are the final values of x after both thread1 and thread2 have run one iteration?

   x=5

   Thread 1 runs to completion, x=6
   Thread 2 runs to completion, x = 12

   Thread 2 runs to completion, x = 10
   Thread 1 runs to completion, x = 11

   Thread 1 loads x and is pre-empted
   Thread 2 loads x and writes back 5 x 2 = 10
   Thread 1 increments x to 6 and writes it back

   Thread 2 loads x and is pre-empted
   Thread 1 loads x, updates it to 6, writes back
   Thread 2 gets 5 x 2 = 10

   Possible values are 12, 11, 6 and 10

c. Research into the term "race condition". How does this program demonstrate race conditions?

   A race condition is a condition where the outcome of a computation that is performed by multiple processes/threads depends on which process/thread finishes first.

d. Which of the answers in b. is correct? How do you define correctness in multithreaded programs?

   Either 12 or 11 depending on the intended sequence of execution.

e. In general, in multithreaded programs shared variables are often updated correctly, and are sometimes updated wrongly. Why?

   Either because the threads are executed in the wrong sequence, or because one thread gets pre-empted before it can save its results and the next thread gets a stale value. The first thread then overwrites the next thread's results.

<u>Question 2</u>

How are local variables created in C (GIYF)? Can local variables be affected by race conditions? Why or why not?

Local variables are created on the process's stack when a function is called. They're popped off and lost when the function exits.

They can never cause race conditions because only the thread or process calling that function can access to the variables. Since no other process or thread has access, race conditions are not possible.

Question 3

Co-operative multitaskers technically cannot suffer from race conditions. Why not? Despite this however, correctness of multithreaded processes in co-operative multitaskers is not guaranteed. Why not?

There are two reasons:

- The sequence of process execution might be wrong. The scheduler might choose to run process A before B, when A depends on a result in B.
- A process may inadvertently give up control of the CPU before completing calculations, causing a dependent process to run and get the wrong results. An example of this is when a process decides to call printf, which will trigger an OS call that might trigger a context switch without the programmer's knowledge.

Question 4

```
#define FALSE  0
#define TRUE   1
#define N      2                    /* number of processes */

int turn;                           /* whose turn is it? */
int interested[N];                  /* all values initially 0 (FALSE) */

void enter_region(int process);     /* process is 0 or 1 */
{
    int other;                      /* number of the other process */

    other = 1 – process;            /* the opposite of process */
    interested[process] = TRUE;     /* show that you are interested */
    turn = process;                 /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)      /* process: who is leaving */
{
    interested[process] = FALSE;    /* indicate departure from critical region */
}
```

i) Explain how Peterson's Solution enforces mutual exclusion.

At first glance Peterson's Solution looks like a "lock variable" solution, but with some key differences:

- There are TWO different lock variables, one for each process.

- Each processes only WRITES to the lock variable, which is an atomic operation. The main weakness of the Lock Variable solution is that it involved read-update-write operations on a single shared variable, which does not happen in this case.
- Before entering the critical section, both processes check to see whether THE OTHER process intends to enter. If so it waits.
- Setting interested to TRUE takes place BEFORE this check, so in the worst case, both set interested to true at the same time, leading to deadlock, which brings us to:

ii) Explain how the “turn” variable in Peterson’s Solution prevents deadlock.

Without “turn” in “enter_region” function, deadlock will occur.

P0	P1
other=1	
Interested[0]=1	
* pre-empted *	other=0;
	Interested[1]=1;
while(interested[1]==1); // Loops forever	* pre-empted *
…	
* pre-empted *	while(interested[0]==1); // Loops forever

Now both P0 and P1 are stuck in the while loops.

Adding in turn will prevent this, because turn will either be 0 or 1. When turn is 0, process 0 is stuck in the loop and process 1 will execute the critical section, and vice versa.

IT5002 Computer Systems and Applications
Tutorial 8 Suggested Solutions

Question 1

It is explained in the lecture. The problem with allocation is that, in order to minimize the meta-data, we allocate memory in blocks of fixed size. As such, in every block, some memory remains unused. This leads to internal fragmentation. Now, when we allocate a large area of memory, this will take several blocks. We need these blocks to be contiguous. It may be the case that our system might have the required number of free blocks that would cover the allocation request, but if these blocks are not contiguous, it is not possible to carry the allocation request through. This is external fragmentation.

Internal fragmentation can be reduced by smaller size of allocation blocks.
External fragmentation can be reduced by relocating occupied blocks.

Question 2

(a) 219+430, legal access
(b) 2300+10, legal access
(c) 1327+500, legal access
(d) 1952+400, illegal access

Question 3

List the blocks in the system

(a) 256 allocated, 256 free, 512 free
(b) 256 allocated, 128 allocated, 128 free, 512 free
(c) 256 allocated, 128 allocated, 64 allocated, 64 free, 512 free
(d) 256 a, 128 a, 64 a, 64 f, 256 a, 256 f
(e) -- first block freed
  256 f, 128 a, 64 a, 64 f, 256 a, 256 f
(f) -- third block freed
  256 f, 128 a, 128 f, 256 a, 256 f
(g) -- fourth block freed
  256 f, 128 a, 128 f, 512 f

IT5002 Computer Systems and Applications
Tutorial 9

1. We consider a simple file system illustrated below:

Partition Information (Free Space Information)

- Bitmap is used, with a total of 32 file data blocks (1 = Free, 0 = Occupied):

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1
16	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
1	0	1	0	1	0	1	0	1	1	1	0	0	1	0	0

Directory Structure + File Information:

- Directory structures are stored in 4 "directory" blocks. Directory entries (both files and subdirectories) of a directory are stored in a single directory block.
- Directory entry:
  - **For File:** Indicates the first and last data block number.
  - **For Subdirectory:** Indicates the directory structure block number that contains the subdirectory's directory entries.
  - The "`/`" root directory has the directory block number 0.

0	**1**	**2**	**3**

`y │Dir │ 3`
`f │File│ 12│ 2`
`x │Dir │ 1`
	`g │File│ 0│ 31`
	`z │Dir │ 2`
		`k │File│ 6│ 6`	`i │File│ 1│ 3`
			`h │File│27│28`

File Data:

- Linked list allocation is used. The first value in the data block is the "next" block pointer, with "-1" to indicate the end of data block.
- Each data block is 1 KB. For simplicity, we show only a couple of letters/numbers in each block.

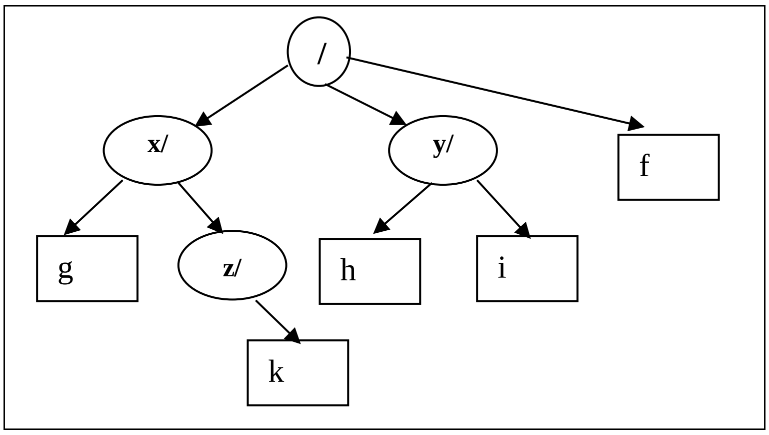
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
11 AL	9 TH	-1 S!	-1 ND	23 GS	-1 SO	-1 :)	10 TE
8	**9**	**10**	**11**	**12**	**13**	**14**	**15**
31 RE	3 EE	28 M:	31 OH	19 SE	13 AH	4 IN	17 NO
16	**17**	**18**	**19**	**20**	**21**	**22**	**23**
30 YE	2 OU	1 ON	17 RI	26 EV	14 AT	21 DA	7 YS
24	**25**	**26**	**27**	**28**	**29**	**30**	**31**
-1 HO	18 ME	0 AL	30 OP	-1 -(	5 LO	21 ER	-1 A!

a. (Basic Info) Give:
   - The current free capacity of the disk.
   - The current user view of the directory structure.
b. (File Paths) Walkthrough the file path checking for:
   - "`/y/i`"
   - "`/x/z/i`"
c. (File access) Access the entire content for the following files:
   - "`/x/z/k`"
   - "`/y/h`"
d. (Create file) Add a new file "`/y/n`" with 5 blocks of content. You can assume we always use the free block with the smallest block number. Indicate **all changes required to add the file.**

ANS:

a.

- **~12KB free space (12 '1' in Bitmap, each data block is 1KB). Due to the linked list file allocation overhead, the actual capacity is a little bit smaller.**



b.

Only the directory block number is indicated:

- `/y/i`: 0, 3 (successful)
- `/x/z/i`: 0, 1, 2 (failed)

c.

- `/x/z/k`: block 6, content = "`:)`"
- `/y/h`: blocks 27, 30, 21, 14, 4, 23, 7, 10, 28, content = "`OPERATINGSYSTEM:-(`"

d.

Bitmap updated: Bit 5, 8, 13, 15, 16 changed to 0
Directory block 3 (for `/y`) updated: "`n |File| 5|16`" added
Data Blocks 5, 8, 13, 15, 16 (next block pointer changed, with -1 in block 16).

2. Let us compare and contrast the various ways of implementing files in this question.

   Below are the hardware parameters:

   a. Number of disk blocks = 2^{16} = 65,536 blocks (i.e. each disk block number = 2 bytes).
   b. Disk block size = 512 bytes.

   Points of comparison:

   - **Total Number of Data Blocks:** Number of disk blocks needed to store the file data.
   - **Overhead**: Extra bookkeeping information in bytes. Focus only on information directly related to keep track of file data. You can ignore the space needed for file name and other metadata for this question.
   - **Worst Case Disk Access:** The worst case number of disk accesses needed to get to a particular block in the file. May include accessing book keeping information if they are in disk. Specify the block which causes the worst case.

   File data allocation schemes under consideration:

   A. Contiguous
   B. Linked list
   C. File allocation table
   D. Index block

   Fill in the following table for the indicated file size. State assumptions for your calculation (if any).

File size: 100 b	**Total Number of Data Blocks**	**Overhead**	**Worst Case Disk Access**
A			
B			
C			
D			

File size: 132,000 b	**Total Number of Data Blocks**	**Overhead**	**Worst Case Disk Access**
A			
B			
C			
D			

File size: 33,554,432 b	**Total Number of Data Blocks**	**Overhead**	**Worst Case Disk Access**
A			

B			
C			
D			

ANS:

Note that with different assumptions, you may need to adjust your answer accordingly.
Assumptions:
For A, the length of blocks is 2 bytes (i.e. can take all disk blocks if needed).
For B, we keep pointers to both first and last block.
For D, we use the "linked" scheme to chain up the index blocks when needed.

Basic Calculation:
For B, one disk block can store 512-2 (the "next" disk block pointer) = 510 bytes only. Although the "next" disk block pointer overhead is already reflected by the larger number of disk blocks required, we still list them explicitly for learning purpose.
For C, the whole FAT cost $2^{16} * 2 = 2^{17}$.
For D, each index block can store 512 / 2 = 256 disk addresses. We use the last disk address to chain to the next index block, i.e. 255 disk addresses can be stored for file data blocks.

File size: 100 b	**Total Number of Data Blocks**	**Overhead**	**Worst Case Disk Access**
A	**1**	**2 (start) + 2 (length)**	**1**
B	**1**	**2 (first) + 2 (last) + 2 (1 pointer)**	**1**
C	**1**	**2^{17} (FAT) + 2 (start)**	**1**
D	**1**	**512 (Index Block) + 2 (Location of Idx Blk)**	**1 (Index block) + 1**

File size: 132,000 b	**Total Number of Data Blocks**	**Overhead**	**Worst Case Disk Access**
A	**258**	**2 (start) + 2 (length)**	**1**
B	**259**	**2 (first) + 2 (last) + 518 (259 pointers)**	**258 access for the 2nd last block.**
C	**258**	**2^{17} (FAT) + 2 (start)**	**1**
D	**258**	**1024 (2 × Index Block) + 2 (Location of Idx Blk)**	**2 (Index block) + 1 for any block beyond 255th.**

File size: 33,554,432 b	**Total Number of Data Blocks**	**Overhead**	**Worst Case Disk Access**
A	**65,536**	**2 (start) + 2 (length)**	**1**

B	**Cannot store (Need 65,794)**	**---**	**---**
C	**65,536**	**2^{17} (FAT) + 2 (start)**	**1**
D	**Cannot store (Need 65,794)**	**---**	**---**

3. We are given the following table of contents in an inode. Suppose that each data block holds 2048 bytes of data. List down the block numbers that would be read/written by the following operations. Assume that fp points to a validly open file with this TOC.

Table of Contents
15
18
12
22

Operation	Block number read/written
fseek(fp, 1121, SEEK_SET)	-
fwrite(buff, sizeof(char), 128, fp)	**15**
fseek(fp, 5523, SEEK_SET)	-
fread(buff, sizeof(char), 256, fp)	**12**
fseek(fp, 8121, SEEK_SET)	-
fwrite(buff, sizeof(int), 25, fp)	**22**