Format Specifiers

Placeholder	Variable Type	Function Use
`%c`	`char`	`printf` / `scanf`
`%d`	`int`	`printf` / `scanf`
`%f`	`float` / `double`	`printf` / `scanf`
`%f`	`float`	`printf`
`%lf` [1]	`double`	`scanf`
`%e`	`float` / `double`	`printf` (*scientific notation*)

```c
#include <stdio.h>

int main(void) {
  char c;
  scanf("%c", &c);
  switch(c) {
    case 'K':
      printf("King\n");
      break;
    case 'Q':
      printf("Queen\n");
      break;
    case 'R':
      printf("Rook\n");
      break;
    case 'B':
      printf("Bishop\n");
      break;
    case 'N':
      printf("Knight\n");
      break;
  }
  return 0;
}
```

Increment/Decrement

Note the order of the increment/decrement and return of value. If the variable `x` has a value of `3`, then after:

Evaluation | Replit

- `y = x++;` → `y` has a value of `3` and `x` has a value of `4`.
- `y = x--;` → `y` has a value of `3` and `x` has a value of `2`.
- `y = ++x;` → `y` has a value of `4` and `x` has a value of `4`.
- `y = --x;` → `y` has a value of `2` and `x` has a value of `2`.

`x++`	Return the value of `x`; **then** increment `x` by `1`
`x--`	Return the value of `x`; **then** decrement `x` by `1`
`++x`	Increment `x` by `1`; **then** return the value of `x`
`--x`	Decrement `x` by `1`; **then** return the value of `x`

Encoding

We can represent numbers in *certain* bases in some programming languages and/or software.

C | Replit | MIPS | Verilog

- **Octal**: Prefix `0`. *e.g.*, `032` represents the octal number $(32)_8$.
- **Hexadecimal**: Prefix `0x`. *e.g.*, `0x32` represents the hexadecimal number $(32)_{16}$.
- **Binary**: Prefix `0b`. *e.g.*, `0b10100` represents the binary number $(10100)_2$.

Non-Standard ›

Type	Meaning	Size	Encoding	Range
`char`	Character/signed byte	8 bits/1 byte	2s complement	-128 to 127
`unsigned char`	Character/unsigned byte	8 bits/1 byte	Unsigned	0 to 255
`short`	Signed short integer	16 bits/2 bytes	2s complement	-32768 to 32767
`unsigned short`	Unsigned short integer	16 bits/2 bytes	Unsigned	0 to 65535
`int`	Signed integer	32 bits/4 bytes	2s complement	-2,147,483,648 to 2,147,483,647
`unsigned int`	Unsigned integer	32 bits/4 bytes	Unsigned	0 to 4,294,967,295
`long` [1]	Signed long integer	64 bits/4 bytes	2s complement	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`unsigned long`	Unsigned long integer	64 bits/4 bytes	Unsigned	0 to 18,446,744,073,709,551,615
`float`	Signed single-precision floating point	32 bits/2 bytes	32-bit IEEE 754	±1.2e-38 to 3.4e+38
`double`	Signed double-precision floating point	64 bits/4 bytes	64-bit IEEE 754	±2.3e-308 to 1.7e+308

(b-1)s and (b)s Complement

Radix Complement

The first extension is to abstract the radix (*i.e.*, the base) into any number. Since we have two kinds of complements (*i.e.*, 1s complement and 2s complement for base 2), the extension to any radix may cause confusion. So, they are actually called ***diminished radix complement*** and ***radix complement*** respectively. Alternatively, they are called (b-1)s complement and (b)s complement respectively.

Number of Digits	Radix	(b-1)s Complement	(b)s Complement
n	b	$-x = b^n-x-1$	$-x = b^n-x$

Example

Consider the number $(43)_{10}$. We can express this in 5-trit[4] number as $(01121)_3$. The negations are:

- ***(b-1)s Complement***: 21101
  a. $-43 = 3^5-43-1 = 199$
  b. $(199)_{10} = (21101)_3$
- ***(b)s Complement***: 21102
  a. $-43 = 3^5-43 = 200$
  b. $(200)_{10} = (21102)_3$

Whole Number of Bits	Fractional Bits	1s Complement	2s Complement
n	f	$-x=2^n-x-2^{-f}$	$-x=2^n-x$
		Invert all bits	Invert all bits Add $\$2^{-f}$

Example

Negate $(5.25)_{10}$ in 1. 1s complement using 4-bit whole number and 2-bit fraction 2. 2s complement using 4-bit whole number and 2-bit fraction

1s Complement | 2s Complement

$(1010.10)_{1s}$

Inversion Method | **Formulaic Method**

1. Find the "equivalent" number: $-5.25 = 2^4-5.25-2^{-2} = 16-5.25-0.25 = 10.5$
2. Find the binary representation of $(10.5)_{10}$: $(1010.10)_2$
3. Change the base: $(1010.10)_{1s}$

Example

Negate $(5.25)_{10}$ in 1. 1s complement using 4-bit whole number and 2-bit fraction 2. 2s complement using 4-bit whole number and 2-bit fraction

1s Complement | **2s Complement**

$(1010.11)_{2s}$

Inversion+Smallest Method | **Formulaic Method**

1. Find the "equivalent" number: $-5.25 = 2^4-5.25-2^{-2} = 10.75$
2. Find the binary representation of $(10.75)_{10}$: $(1010.11)_2$

Common Mistake

What is wrong with the following code?

Code | Replit

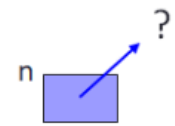
CommonMistake.c

```c
int *n;
*n = 123;
printf("%d\n", *n);
```

Unfortunately, the "Replit" tab does not show the error because it is compiled with Clang that is doing a smart initialisation. Try the following:

1. Click the "Replit" tab, and the the "Shell" tab.
2. Compile with GCC using `gcc main.c`
3. Execute the code using `./a.out`

You should see `Segmentation fault (core dumped)`. This is caused because the pointer variable `n` is not pointing to any valid variable! In fact, if you draw the box-and-arrow diagram, you will not know where to connect the arrow to. If you run this on your own computer, remove the file `core` from your directory as it takes up a lot of space.



Initialisation

Code | Replit

ArrInit.c

```c
// a[0]=54, a[1]=9, a[2]=10
int a[3] = {54, 9, 10};

// size of b is 3 with b[0]=1, b[1]=2, b[2]=3
int b[] = {1, 2, 3};

// c[0]=17, c[1]=3, c[2]=10, c[3]=0, c[4]=0
int c[5] = {17, 3, 10};

// size of d is 2 with d[0]=1, d[1]=2
int d[2] = {1, 2, 3};  // warning issued: excess elements
```

Under Initialisation

Under initialisation happens when you are initialising the array with fewer elements that the size can contain. What will happen is that the rest of the element will be assigned the value of 0. This is independent of the compiler used (GCC or Clang).

After Initialisation

Code | Replit

AfterInit.c

```c
int e[5];
e[5] = {8, 23, 12, -3, 6}; // too late to do this;
                           // compilation error
```

Array Assignment

Code | Replit

ArrayAssignment.c

```c
#define N 10
int source[N] = { 10, 20, 30, 40, 50 };
int dest[N];
dest = source;  // illegal!
                // We cannot assign to the array (i.e., `<array name> = <expr>;`)!
```

Retrieval ≈ Dereferencing

`arr[idx] == *(arr + idx)`

Commutativity

Due to the commutativity of the `+` operator[2], we can have the following weird operation:

Commutativity

```
   arr[idx]
=> *(arr + idx)  // by dereferencing
=> *(idx + arr)  // by commutativity
=> idx[arr]
```

But remember, `idx` is an unsigned `int` and `arr` is an array. Basically, what we want to say is that the following code *weirdly* works.

Code | Replit

CommutativeArr.c

```c
int arr[3] = {1, 2, 3};
printf("%d\n", 1[arr]); // equivalent to arr[1]
```

A String

The following arrays are strings:

1. `char code[7] = {'c', 's', '2', '1', '0', '0', '\0'}`
2. `char name[] = {'C','o','m','p',' ','O','r','g',0}`
3. `char who[] = {65, 100, 105, 0}`

Note the use of integer 0 on the second example as opposed to the null character `'\0'` in the first example. This is accepted because of implicit conversion from 4 bytes `int` to 1 byte `char` since they interchangeable for small integers. This is taken to the extreme in the third example.

Not A String

The following arrays are not strings:

1. `char code[7] = {'c', 's', '1', '0', '1', '0', 'e'}`
2. `int name[] = {'C','o','m','p',' ','O','r','g',0}`
3. `int who[] = {65, 100, 105}`

The first example violates the second rule because it does not end with a null character. The second example violates the first rule because it is not an array of characters. We take this to the extreme again with the third example that violates both rules.

Word Align Check

Questions | Answer

How do we quickly check whether a given memory address is word-aligned or not?

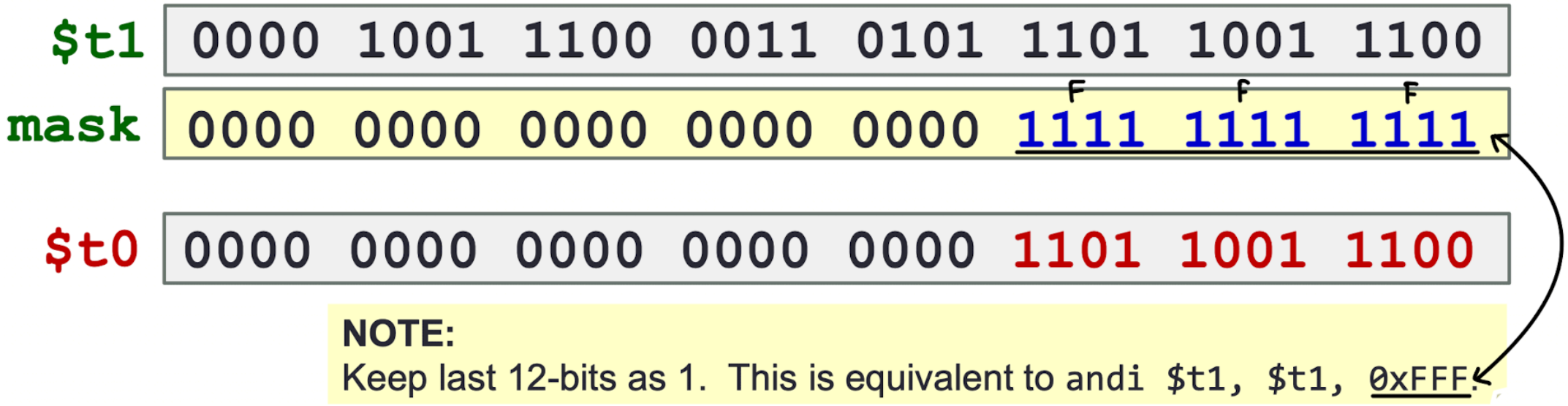
Word Align Check

Questions | Answer

We check if the value of the address *modulo* word size is equal to 0. If a word is 2^n bytes, we can use the bitwise AND operation to bitmask everything ***except*** the last *n*-bits. We then check if the last *n*-bits are all 0s.

5.9 Exercise: **Bitwise AND**

- We are interested in the last 12 bits of the word in register **`$t1`**. Result to be stored in **`$t0`**.
  - **Q:** What's the mask to use?



Notes:

The **`and`** instruction has an immediate version, **`andi`**

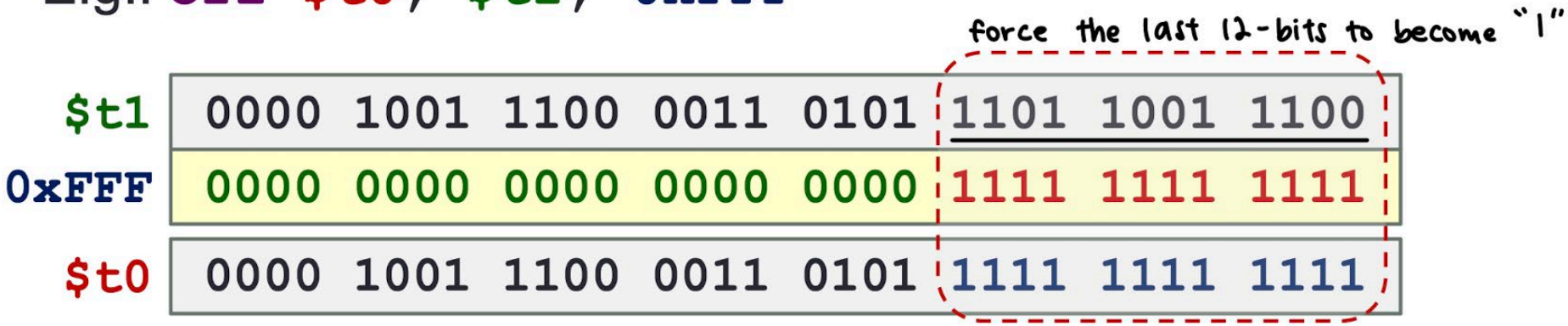
5.10 Logical Operations: **Bitwise OR**

Opcode: `or` (bitwise OR)

Bitwise operation that places a 1 in the result if either operand bit is 1

Example: `or $t0, $t1, $t2`

- The `or` instruction has an immediate version `ori`
- Can be used to force certain bits to 1s
- E.g.: `ori $t0, $t1, 0xFFF`



For `ori $t0, $t1, 0xFFFF` will the upper 16-bits be all 0s or all 1s?

Answer: all 0s (***in other words, this is not sign-extended***)

Naive Translation

Unaligned Load Word | Unaligned Store Word

```
lw $s0, 3($s1)
```

Unaligned Load Word

```
1   lb  $t0, 3($s1)    # 0x00000033
2
3   lb  $t1, 4($s1)    # 0x00000044
4   sll $t1, $t1, 8    # 0x00004400
5   or  $t0, $t0, $t1  # 0x00004433
6
7   lb  $t1, 5($s1)    # 0x00000055
8   sll $t1, $t1, 16   # 0x00550000
9   or  $t0, $t0, $t1  # 0x00554433
10
11  lb  $t1, 6($s1)    # 0x00000066
12  sll $t1, $t1, 24   # 0x66000000
13  or  $t0, $t0, $t1  # 0x66554433
```

Naive Translation

Unaligned Load Word | Unaligned Store Word

```
sw $s0, 3($s1)
```

Unaligned Load Word

```
1   add $t0, zero, $s0 # 0x66554433
2   lb  $t0, 3($s1)    # 0x00000033
3
4   srl $t0, $t0, 8    # 0x00665544
5   lb  $t0, 4($s1)    # 0x00000044
6
7   srl $t0, $t0, 8    # 0x00006655
8   lb  $t0, 4($s1)    # 0x00000055
9
10  srl $t0, $t0, 8    # 0x00000066
11  lb  $t0, 4($s1)    # 0x00000066
```

blt

Pseudo-Instruction Translation

blt bgt ble bge

`blt $rs, $rt, label`

We do the translation using an intermediate form: `if (($rs < $rt) != 0) goto label`

blt Translation

```
1  slt $t0, $rs, $rt
2  bne $t0, $zero, L
```

bgt

Pseudo-Instruction Translation

blt **bgt** ble bge

`bgt $rs, $rt, label`

We do the translation using an intermediate form: `if (($rt < $rs) != 0) goto label`

bgt Translation

```
1  slt $t0, $rt, $rs
2  bne $t0, $zero, L
```

ble

Pseudo-Instruction Translation

blt bgt **ble** bge

`ble $rs, $rt, label`

We do the translation using an intermediate form: `if (($rt < $rs) == 0) goto label`

ble Translation

```
1  slt $t0, $rt, $rs
2  beq $t0, $zero, L
```

bge

Pseudo-Instruction Translation

blt bgt ble **bge**

`bge $rs, $rt, label`

We do the translation using an intermediate form: `if (($rs < $rt) == 0) goto label`

blt Translation

```
1  slt $t0, $rs, $rt
2  beq $t0, $zero, L
```

Translation Idea

Here, the comparison `$rs < $rt` will be 1 if the condition is true. So we simply check that the result is not equal to 0. Then we can evaluate `$rs > $rt` using `slt` by swapping the two registers! In other words:

```
$rs > $rt
≡ $rt < $rs
```

For the `ble` we simply do the negation.

```
$rs <= $rt
≡ !($rs > $rt)
≡ !($rt < $rs)
≡ ($rt < $rs) == 0
```

Lastly, for the `bge` we do the same trick as before again.

```
$rs >= $rt
≡ $rt <= $rs
≡ !($rt > $rs)
≡ !($rs < $rt)
≡ ($rs < $rt) == 0
```

Array Indexing

Version 1 Version 2

In this version, we use the following simple C code:

Count Zeroes V1 (in C)

```c
result = 0;
i = 0;
while (i < 40) {   // How to compare correctly?
  if (A[i] == 0) { // How to translate A[i] correctly?
    result++;
  }
  i++;
}
```

Since we use a temporary variable `i`, we need to map this. Let us map this to `$t1`. We also need to store the constant `40` for comparison, we will use `$t2`. Lastly, note that we need to compute the correct address. Since this is an integer array, it will be word aligned. Which means, the actual offset will need to be multiplied by 4. We use both `$t3` and `$t4` to compute the actual address for load.

Count Zeroes V1 (in MIPS)

```
      addi $t8, $zero, 0
      addi $t1, $zero, 0
      addi $t2, $zero, 40    # end point
loop: bge  $t1, $t2, end     # pseudo-instruction, can you use beq?
      sll  $t3, $t1, 2       # $t3 = i*4     (for offset computation)
      add  $t4, $t0, $t3     # $t4 = &A[$t3] (after offset computation)
      lw   $t5, 0($t4)       # $t5 = A[$t3]  (deferencing using *$t4)
      bne  $t5, $zero, skip
      addi $t8, $t8, 1       # result++
skip: addi $t1, $t1, 1       # i++
      j loop
end:
```

Translation Idea

Here, the comparison `$rs < $rt` will be 1 if the condition is true. So we simply check that the result is not equal to 0. Then we can evaluate `$rs > $rt` using `slt` by swapping the two registers! In other words:

```
$rs > $rt
≡ $rt < $rs
```

For the `ble` we simply do the negation.

```
$rs <= $rt
≡ !($rs > $rt)
≡ !($rt < $rs)
≡ ($rt < $rs) == 0
```

Lastly, for the `bge` we do the same trick as before again.

```
$rs >= $rt
≡ $rt <= $rs
≡ !($rt > $rs)
≡ !($rs < $rt)
≡ ($rs < $rt) == 0
```

Pointer Arithmetic

Version 1 | Version 2

In this version, we use pointer arithmetic version of the C code:

Count Zeroes V2 (in C)

```c
result = 0;
curr = &A[0];
last = &A[40];
while (curr < last) { // How to compare correctly?
  if (*curr == 0) {   // How to translate curr correctly?
    result++;
  }
  curr++;
}
```

Here we use two additional variables `curr` and `last`. `curr` is address of the current item, which we will be incremented using pointer arithmetic. `last` is a computed address of the last element. We can compare address as if they are numbers because MIPS register has no data type. However, this omission of data type produce a problem for `curr++`. This should increment the address by 4 because this is an integer array. So remember to do this on your MIPS code. Now, since `curr` is already the address, we do not need to do the multiplication by 4 and add the base address. Instead, we can simply load.

Count Zeroes V2 (in MIPS)

```
      addi $t8, $zero, 0
      addi $t1, $t0, 0      # curr = &A[0];
      addi $t2, $t0, 160    # last = &A[40];
loop: bge  $t1, $t2, end    # address comparison! can you use beq?
      lw   $t3, 0($t1)      # $t3 = *curr  (simple load)
      bne  $t3, $zero, skip
      addi $t8, $t8, 1      # result++
skip: addi $t1, $t1, 4      # curr++ (move to next item, but add 4 to address)
      j loop
end:
```

Branch further away

Branch

Given the instruction `beq $s0, $s1, label`, what happen if the address for `label` is farther away from the `$PC` than what can be supported by `beq` (*or* *bne*) instruction? Here, we can actually chain `beq` (*or* *bne*) with `j` instruction. In other words, we introduce an intermediate label (*e.g.*, `mid_branch`) that contains a `j label` instruction. We then replace the original instruction with `beq $s0, $s1, mid_branch`.

Of course, you can also chain with another `beq` instruction instead of `j` instruction. This will be especially useful when you are trying to branch to outside of the 256MB boundary, which we will discuss later.

Original

```
label: # can be above or below
  :
  :    # code omitted
  :
beq $s0, $s1, label
```

Replacement

```
label: # can be above or below
  :
  :    # code omitted
  :
mid_branch: j label
  :
  :    # code omitted
  :
beq $s0, $s1, mid_branch
```

Jump further away

Jump

Given the instruction `j label`, what happen if the address of `label` is outside of the 256MB boundary? In this case, we will need to chain the instruction with `beq` instruction that is guaranteed to always branch. Similar to before, we add an intermediate label. Here, the `beq` will need to be at or near the edge of the 256MB boundary.

Original

```
label: # can be above or below
  :
  :    # code omitted <- boundary
  :
j label
```

Replacement

```
label: # can be above or below
  :
  :    # code omitted <- boundary
  :
mid_jump: beq $s0, $s1, label
  :
  :    # code omitted
  :
j mid_jump
```

Warning when Jumping far

Warning

There are a few things we have to be careful of. Firstly, we need to ensure that the added instruction can be reached from the original instruction. Additionally, we need to ensure that the added instruction can reach the original target. Since we are adding instructions, this may mean that the original instruction is pushed down which may change the boundary. This also means that we may need to change the immediate value for other instructions.

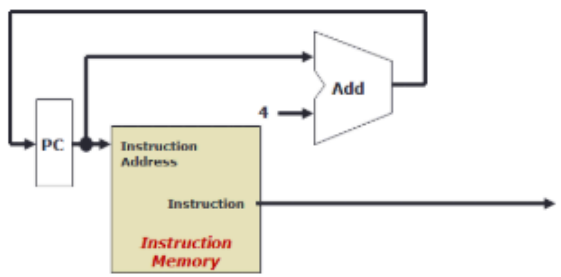
Secondly, we need to ensure that the added instruction is not executed unintentionally. In our replacement recipe above, we kind of omitted the code before and after the added instruction. One trick here is that we can add branch/jump before the added instruction to branch to the line immediately after the added instruction. If we do that, then we ensure that normal execution of the program will ignore this added instruction.

Replacement

```
  :    # code omitted <- boundary
          beq $zero, $zero, after  # ---+
mid_jump: # added instruction           |
after:                             # <--+
  :    # code omitted
```

	Big-Endian	Little-Endian
Order	Most significant byte stored in lowest address (big-end)	Least significant byte stored in lowest address (little-end)
Processor	IBM 360/370, Motorola 6800, MIPS, SPARC	Intel 80x86, DEC VAX, DEC Alpha
Example	0xDE AD BE EF is stored as 0 DE 1 AD 2 BE 3 EF	0xDE AD BE EF is stored as 0 EF 1 BE 2 AD 3 DE

Clock



There is a slight potential problem when we try to combine "Step 1" and "Step 2" above. In "Step 1", we are reading the value of `$PC` but in "Step 2" we are updating the value of `$PC`. How can we do **both** at the same time? How can it works properly? The solution is to use a clock.

We simply define a clock as "something" that repeats at a regular interval. In the context of processor and circuit, it can be abstracted into a regular signal as seen on the right. Here, you see that the arrow up appears at regular time interval.

There are several convention about a clock. The convention we use for this current implementation is to read the value of `$PC` during the first half of the clock period and update the value of `$PC` with `$PC` + 4 at the **next rising clock edge**.

ALU Control Unit

Let us first summarise the formula for each `ALUcontrol` bit.

ALUcontrol

```
ALUcontrol3 = 0
ALUcontrol2 = (F1 . ALUop1) + (ALUop0)
ALUcontrol1 = !ALUop1 + !F2
ALUcontrol0 = (F0 + F3) . ALUop1
```

EXPONENT	MANTISA	VALUE
0	0	exact 0
255	0	Infinity
0	not 0	denormalised
255	not 0	Not a number (NAN)

4 bit Excess-N

To ensure even distribution between positive and negative numbers, we can either choose Excess-7 or Excess-8 representation. The most common one is Excess-8.

Excess-8 | Excess-7

Excess-8	Value	Excess-8	Value
`0000`	-8	`1000`	0
`0001`	-7	`1001`	1
`0010`	-6	`1010`	2
`0011`	-5	`1011`	3
`0100`	-4	`1100`	4
`0101`	-3	`1101`	5
`0110`	-2	`1110`	6
`0111`	-1	`1111`	7

4 bit Excess-N

To ensure even distribution between positive and negative numbers, we can either choose Excess-7 or Excess-8 representation. The most common one is Excess-8.

Excess-8 | Excess-7

Excess-7	Value	Excess-7	Value
`0000`	-7	`1000`	1
`0001`	-6	`1001`	2
`0010`	-5	`1010`	3
`0011`	-4	`1011`	4
`0100`	-3	`1100`	5
`0101`	-2	`1101`	6
`0110`	-1	`1110`	7
`0111`	0	`1111`	8

Consider a 36-bit fixed length instructions with 3 types of instructions:

1. **Type-A**: 3 operands (2 addresses + 1 register number)
2. **Type-B**: 2 operands (1 address + 1 register number)
3. **Type-C**: 0 operand

The operands take the following number of bits:

- **Address**: 15 bits.
- **Register Number**: 3 bits.

Additionally, you are asked that each type of instructions must have *at least* the following number of instructions:

1. **Type-A**: 7 instructions
2. **Type-B**: 500 instructions
3. **Type-C**: 50 instructions

Answer Steps

One possible answer is as follows:

Types	3 Bits	15 Bits	15 Bits	3 Bits
A	`opcode`	`Address`	`Address`	`Register`
B	`opcode`	`opcode2`	`Address`	`Register`
C	`opcode`	`opcode2`	*unused*	*unused*

More specifically, we can specify the following `opcode` and `opcode2`:

Types	3 Bits	15 Bits
A	`000` - `110`	
B	`111`	`000000 000000000` - `000000 111111111` ($2^9 = 512$)
C	`111`	`000001 000000000` - `111111 000000000` (2^6-1 = 63)

Q4 ISA Instruction Counts

Observations:

1. Opcode in Class B shouldn't be a prefix of Class A

2. The presence of each Class B opcode eliminates $2^{11-6} = 32$ Class A opcode

There are $2^{11} = 2048$ Class A opcodes theoretically.

Min:

63 Class B, eliminating 63 x 32 = 2016 Class A opcodes

Total: 63 + (2048 – 2016) = 95

Class A:

11 bit 5 bit

Class B:

6 bit 5 bit 5 bit

Q4 ISA Instruction Counts

Observations:

1. Opcode in Class B shouldn't be a prefix of Class A

2. The presence of each Class B opcode eliminates $2^{11-6} = 32$ Class A opcode

There are $2^{11} = 2048$ Class A opcodes theoretically.

Max:

1 Class B, eliminating 1 x 32 = 32 Class A opcodes.

Total: 1 + (2048 – 32) = 2017

Class A:

11 bit 5 bit

Class B:

6 bit 5 bit 5 bit

Backup:

Type A: 4-bit opcode

Type B: 8-bit opcode

Type C: 10-bit opcode

Find min and max number of instructions (with at least one instruction of each type).

Min: addition

Max: subtraction

Min:

Maximize A, then maximize B, leave the rest to C

A: $2^4 - 1 = 15$, leave one 4-bit prefix to B and C

B: $1 \times 2^{8-4} - 1 = 15$, leave one 8-bit prefix to C

C: $1 \times 2^{10-8} = 4$

Total: $15 + 15 + 4 = 34$

Max:

Minimize A, then minimize B, leave the rest of C

A: 1

B: 1

Each type A eliminates $2^{8-4} = 16$ type B, and $2^{10-4} = 64$ type C

Each type B eliminates $2^{10-8} = 4$ type C

C: $2^{10} - 1 \times 64 - 1 \times 4 = 956$

Total: $1 + 1 + 956 = 958$

Variable mapping: *a* → $s0; *b* → $s1; *c* → $s2

Q2. (a) Set bits 2, 8, 9, 14 and 16 of *b* to 1. Leave the other bits unchanged.

Recall:
x OR 0 = *x*
x OR 1 = 1

Example: Before

b = 0011 0000 0111 0000 0110 1101 0010 0001.

(Bits 2, 8, 9, 14 and 16 are underlined.)

After

b = 0011 0000 0111 0001 0110 1111 0010 0101.

Conceptually:

```
ori  $s1, $s1, 0b0000 0000 0000 0001 0100 0011 0000 0100
```

```
lui  $t0, 0b1                                  # set bit 16.
ori $t0, $t0, 0b0100 0011 0000 0100 # set bits 14,9,8,2.
or  $s1, $s1, $t0
```

Variable mapping: *a* → $s0; *b* → $s1; *c* → $s2

Q2. (b) Copy over bits 1, 3 and 7 of *b* into *a*, without changing any other bits of *a*.

Recall:
x AND 0 = 0
x AND 1 = *x*

Recall:
x OR 0 = *x*
x OR 1 = 1

Example: Before (assume that the most significant 24 bits are all zeroes)

a = 0 0 … 0 0 0 0 1 0 1 0 1 0.

b = 0 0 … 0 0 1 1 0 1 1 1 0 0.

After

a = 0 0 … 0 0 1 0 1 0 1 0 0 0.

b = 0 0 … 0 0 1 1 0 1 1 1 0 0.

```
andi  $t0, $s1, 0b0000 0000 1000 1010
lui   $t1, 0b1111 1111 1111 1111
ori   $t1, $t1, 0b1111 1111 0111 0101
and   $s0, $s0, $t1
or    $s0, $s0, $t0
```

1. Get bits 1, 3 and 7 of b
2. Get the rest of bits of a
   - Use a 32-bit mask
3. Combine them together

Variable mapping: *a* → $s0; *b* → $s1; *c* → $s2

Q2. (c) Make bits 2, 4 and 8 of *c* the inverse of bits 1, 3 and 7 of *b*.

Recall:
x XOR 0 = *x*
x XOR 1 = *x*′

Example: Before (assume that the … part are all zeroes)

b = 0 0 … 0 0 1 1 0 1 1 1 0 0.

c = 0 0 … 0 1 0 0 1 0 1 0 1 0.

After

b = 0 0 … 0 0 1 1 0 1 1 1 0 0.

c = 0 0 … 0 0 0 0 1 0 1 1 1 0.

```
xori  $t0, $s1, 0b1000 1010
andi $t0, $t0, 0b1000 1010
sll    $t0, $t0, 1
lui    $t1, 0b1111 1111 1111 1111
ori    $t1, $t1, 0b1111 1110 1110 1011
and   $s2, $s2, $t1
or     $s2, $s2, $t0
```

1. Prepare b
   1. Negate bits 1, 3 and 7 of b
   2. Set other bits to 0
   3. Left shift by 1 position to bits 2, 4 and 8
2. Prepare c
   1. Get all bits of c except bits 2, 4 and 8
      - Use a 32-bit mask
3. Combine them together

SUB instruction

Inst-Mem (400) → Reg.File (200) → MUX (ALUSrc) (30) → ALU (120) → MUX (MToR) (30) → Reg.File (200)

Control (100)

Not critical path

Q2(a)

Inst-Mem
400ps

Adder
100ps

MUX
30ps

ALU
120ps

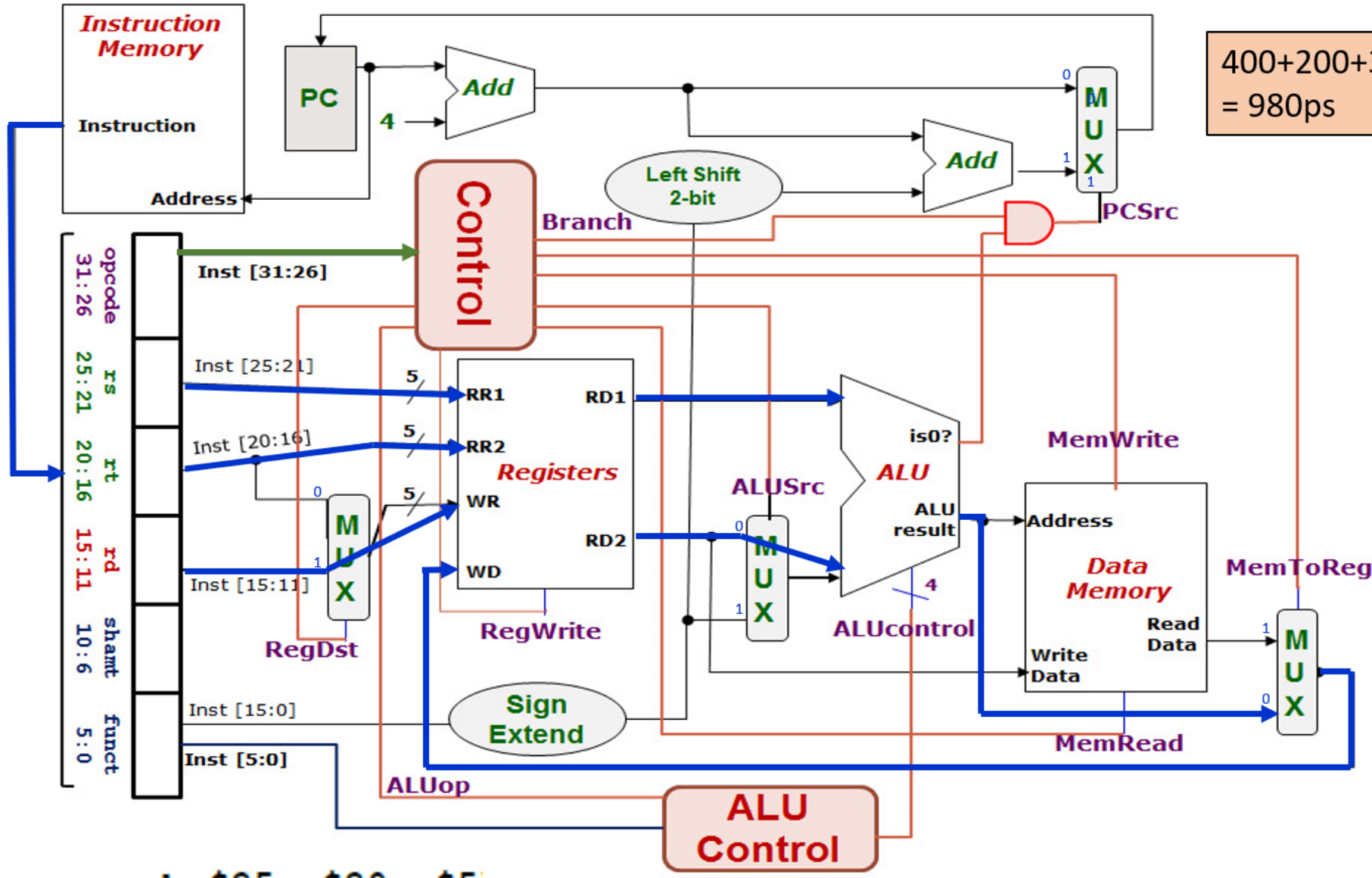
Reg-File
200ps

Data-Mem
350ps

Control/ALU Control
100ps

Lshft/signext /AND
20ps

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




```
sub $25, $20, $5
```

LW instruction

Inst-Mem (400) → Reg.File (200) → ALU (120) → DataMem (350) → MUX (MToR) (30) → Reg.File (200)

Inst-Mem (400) → Control (100)

Not critical path

Q2(b)

Inst-Mem
400ps

Adder
100ps

MUX
30ps

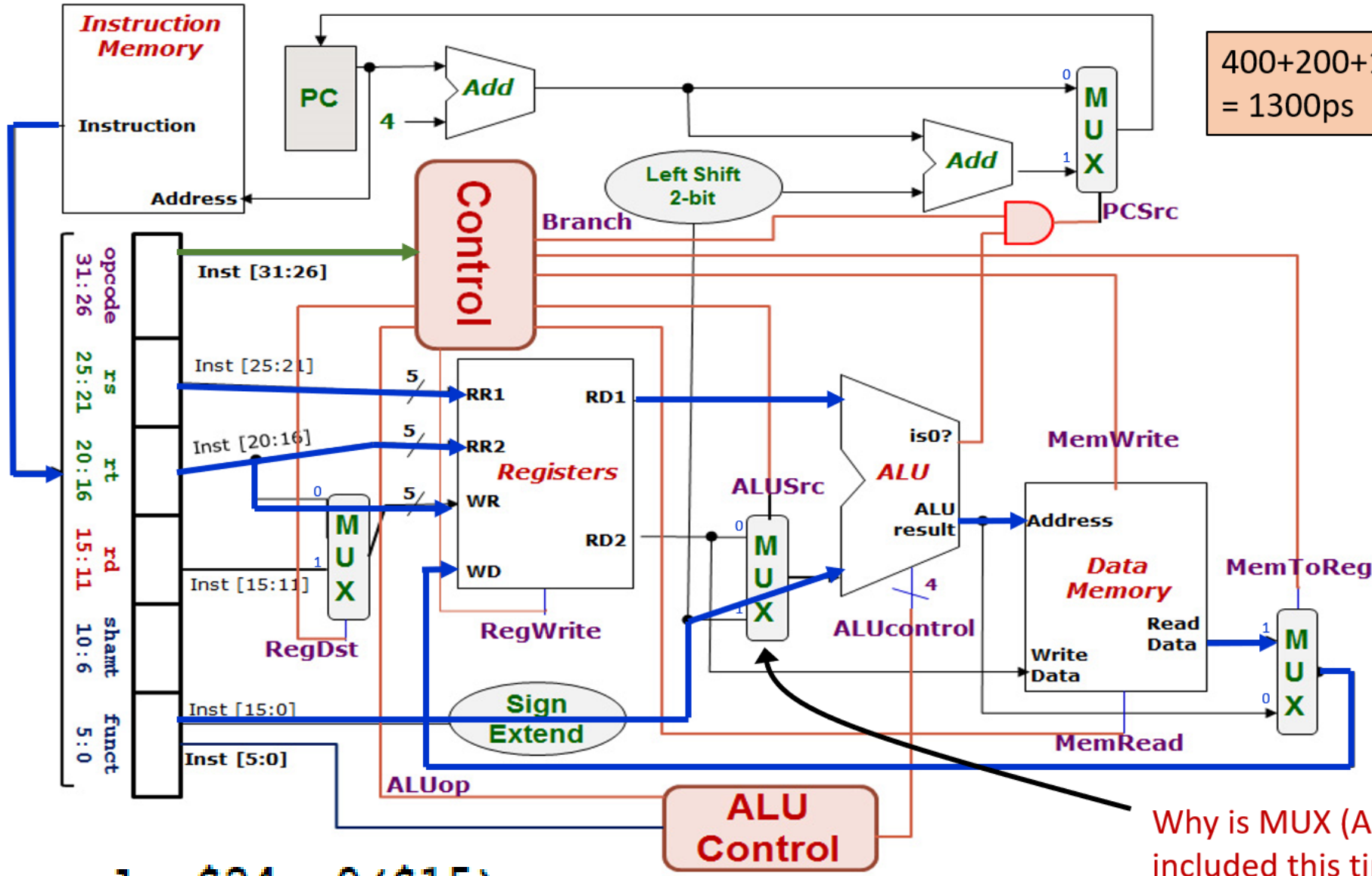
ALU
120ps

Reg-File
200ps

Data-Mem
350ps

Control/ALU Control
100ps

Lshft/signext /AND
20ps



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

Why is MUX (ALUSrc) not included this time?

```
lw $24, 0($15)
```

BEQ instruction

Inst-Mem (400) → Reg.File (200) → MUX (ALUSrc) (30) → ALU (120) → AND (20) → MUX (PCSrc) (30)

Inst-Mem (400) → Control (100)

Not critical path

Q2(c)

Inst-Mem
400ps

Adder
100ps

MUX
30ps

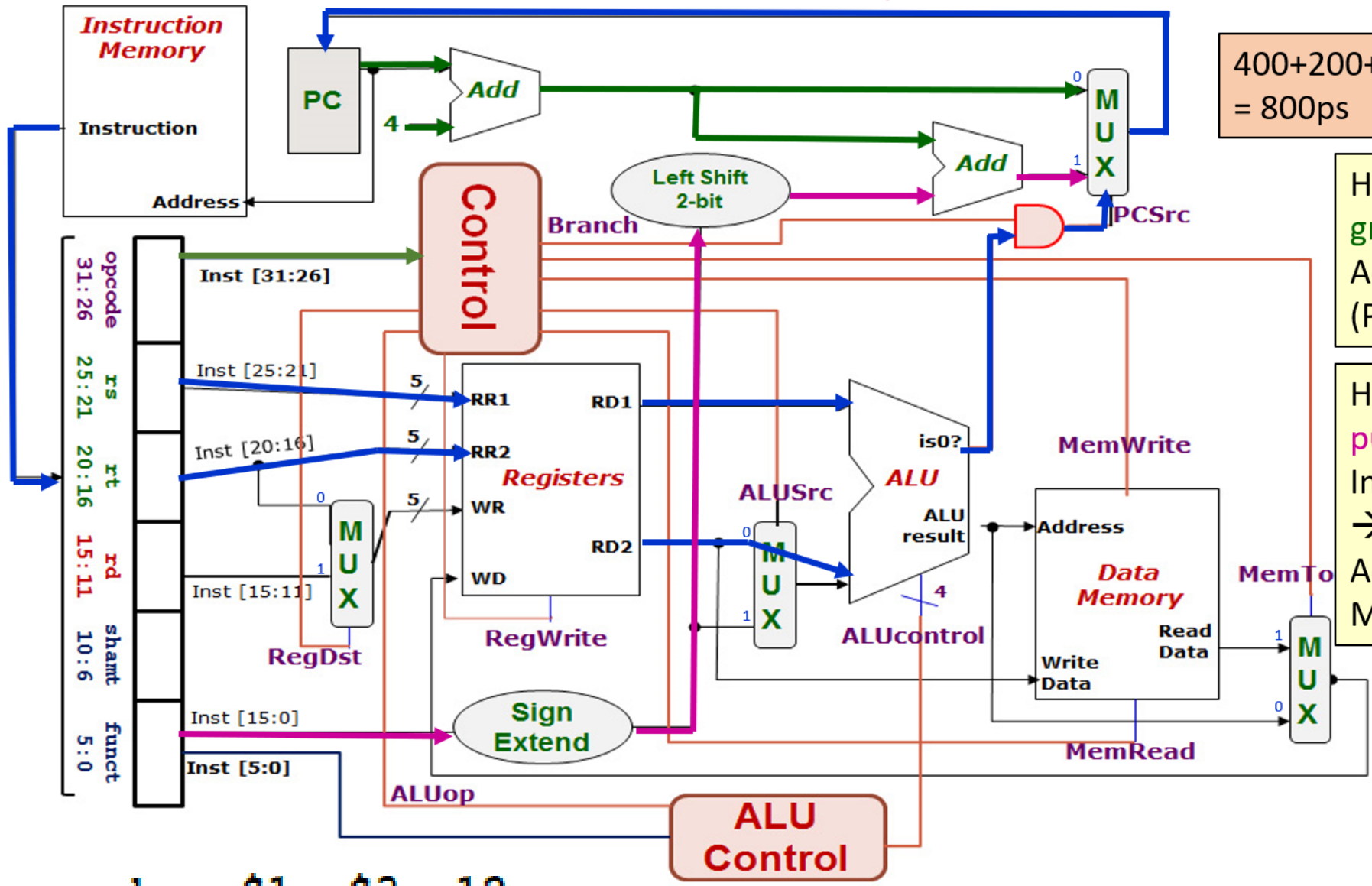
ALU
120ps

Reg-File
200ps

Data-Mem
350ps

Control/ALU Control
100ps

Lshft/signext /AND
20ps



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

How about the green path: PC → Adder → MUX (PCSrc)?

How about the purple path: Inst.Mem → SignExt → LeftShift → Adder → MUX(PCSrc)?

```
beq $1, $3, 12
```