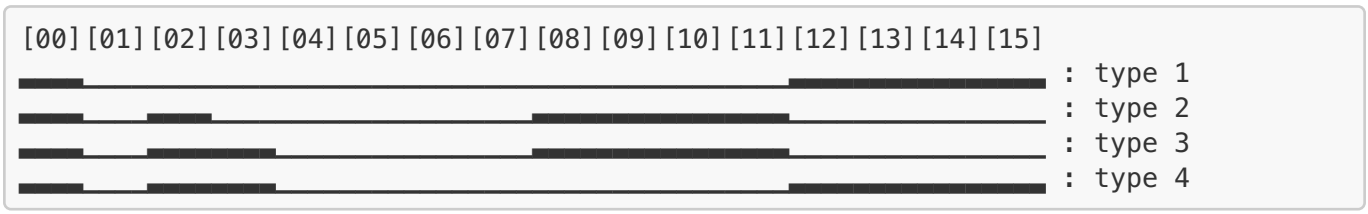
3 Processor Design

3.1 Instruction Set Structure

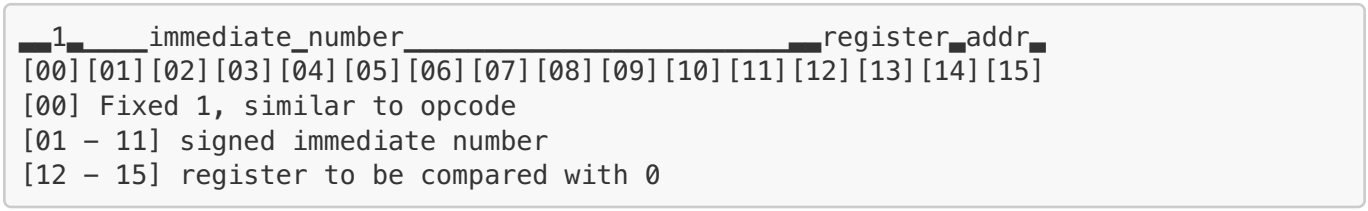
Instruction Definition

Imporant note: All numbers are signed



Different thickness represents separate bit usage space.

Type 1 for conditional branch (bne in assembly)

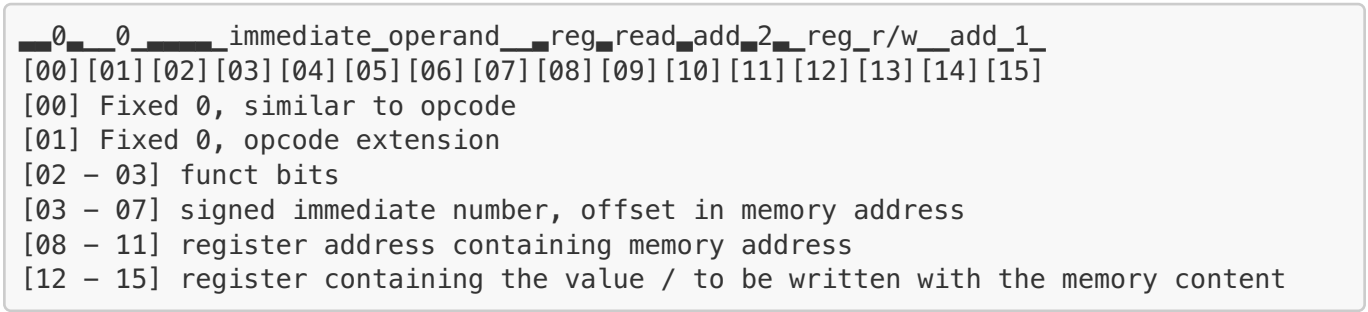


This instruction compares the register value with 0. On finding a non-zero value, the program counter will be set to current PC + imm_number.

E.g.

If register No. 3 contains a non-zero value, the program counter will increase by 4.

Type 2 for load and store



This instruction accesses (reads/writes) memory address reg_r/w_add_1 * 4.

[02] could be 0 or 1 to indicate load or store respectively.

When in load mode, [12-15] is the write register address. When in store mode, [12-15] is the read register address.

E.g.

This instruction copies memory content at (0x10 + $r4 value) to $r3.

```
0 0 1 | 0 1 0 0 0 | 1 1 0 0 | 0 1 0 1
```

This instruction copies the value stored in register 5 to memory at address (0x20 + $r12 value).

Type 3 for 3 arithmetic/logic instruction operating on 3 register operands.

```
▬0▬1▬ope▬n.▬reg_write_add▬▬reg▬read▬add▬2▬reg_read_add_1▬
[00][01][02][03][04][05][06][07][08][09][10][11][12][13][14][15]
[00] Fixed 0
[01] Fixed 1
[02 - 03] Operation distinction bits
[04 - 07] register to write the operation result
[08 - 11] register containing operand 2
[12 - 15] register containing operand 1
```

[02][03] could be 0 0, 0 1, 1 0 to indicate 3 a/l instruction.

- 0 0 for addition,
- 0 1 for left shifting (shifting reg_read_1 to the left by reg_read_2 bits, store in reg_write),
- and 1 0 for logic XOR.

Example: Addition :

```
0 1 | 0 0 | 0 0 1 1 | 0 0 1 0 | 0 0 0 1
```

is equivalent to write in assembly format Add $1, $2, $3 (register order is inversed from the binary).

Say $1 is +10(decimal), $2 is -4, $3 is whatever number, after the operation, $3 should contain +6.

Example: Left shifting :

```
0 1 | 0 1 | 0 0 1 1 | 0 0 1 0 | 0 0 0 1
```

is equivalent to write in assembly format Sll $1, $2, $3

Say $1 is +10(decimal), 0 [...] 1 0 1 0 (binary), $2 is -2, $3 is whatever number, after the operation, $3 should contain +2(decimal).

Type 4 for copy instruction containing 1 register address and a 8-bit immediate number.

```
▬0▬1▬▬1▬▬1▬reg_write_add▬▬▬▬immediate▬number▬▬▬▬▬▬▬▬
[00][01][02][03][04][05][06][07][08][09][10][11][12][13][14][15]
00] Fixed 0
[01] Fixed 1
[02 - 03] Operation distinction bits, fixed to 11
[04 - 07] Register to write the operation result
[08 - 15] Signed immediate number
```

[02][03] set to 1 1 to indicate that it is a copy (Cpy in assembly).

E.g.

```
0 1 1 1 | 0 0 1 0 | 1 0 0 0 1 0 1 1
```

This instruction sets register 2 with a value of -117.

Note that type 3 and type 4 are variations of type 3, while varying [02][03] to allocate the following bit space differently.

Also, all the immediate numbers are signed, to no longer require instructions for signed/unsigned immediate numbers to economize the function space bits.

Negative numbers are the complementary of the positive number + 1.

Instruction format

- I-format : type 2
- J-format : type 1
- R-format : type 3 and 4

Loading 65535 into a register

Suppose initially all the registers are null.

1. Cpy $0, +127 ($0 = +127)

```
0 1 1 1 | 0 0 0 0 | 0 1 1 1 1 1 1 1
```

2. Cpy $1, +7 ($1 = +7)

```
0 1 1 1 | 0 0 0 1 | 0 0 0 0 0 1 1 1
```

3. Sll $0, $1, $2 ($2 = 16256)

```
0 1 0 1 | 0 0 1 0 | 0 0 0 1 | 0 0 0 0
```

4. Add $2, $0, $3 ($3 = 16383)

```
0 1 0 0 | 0 0 1 1 0 0 0 0 | 0 0 1 0
```

5. Cpy $1, +2 ($1 = +2)

```
0 1 1 1 | 0 0 0 1 | 0 0 0 0 0 0 1 0
```

6. Sll $3, $1, $4 ($4 = 65532)

```
0 1 0 1 | 0 1 0 0 | 0 0 0 1 | 0 1 0 0
```

7. Cpy $0, +3 ($0 = +3)

```
0 1 1 1 | 0 0 0 0 | 0 0 0 0 0 0 1 1
```

8. Add $4, $0, $15 ($15 = 65535)

```
0 1 0 0 | 1 1 1 1 | 0 0 0 0 | 0 1 0 0
```

9. PC now points to an instruction that does nothing important, if the branch is not taken, PC detects that instructions have run out, and the program finishes.

Translating the code.

Cpy $0, +32 --- int *a = 0x20

Cpy $1, +0 --- int s = 0 (to ensure that the register value is 0)

Cpy $2, +0 --- int i = 0 (to ensure that the register value is 0)

Cpy $3, +9 --- ($3 = +9, +9 = 10 - 1, given the special position of the conditional branch in the instruction)

Cpy $12, +4 --- ($12 = +4)

Cpy $11, +1 --- ($11 = +1)

XOR $2, $3, $14 --- (compare i and 9, storing the result in $14)

Load $0, $13, +0 --- ($13 = *a)

Add $1, $13, $1 --- s += *a (since $13 = *a)

Add $0, $12, $0 --- a += 4

Add $2, $11, $2 --- i++

Bne $14, -24 (jump back to XOR $2, $3, $14)

Cpy $7, +0 (a placeholder, which has no influence on the program)

3.2 Pipelining

Diagram

Hazards

Data Hazards

Assumed by the project document, values written in the EX stage are immediately available in the ID stage,

thus there's no data hazard.

Control Hazards

In this design, conditional branches are executed in the ID stage, thus if a branch is taken we need to flush one instruction that is currently in IF stage.

Therefore, control hazards do exist.

Structural Hazards

Regarding to register file, we assume that registers are read at the end of the ID stage and written at the beginning of the EX stage, thus no reading and writing contention would happen. The processor implements a writing enabler and a unique writing address, thus no conflict would happen. Meanwhile reading the same register would not do any harm.

Regarding to data memory, memory uses one bit input to set READ/WRITE mode, so accessing memory is safe. In conclusion, we have no structural hazards.

Forwarding

This processor has only three stages: IF, ID and EX.

And as demonstrated above, we have no data hazards, thus at least the register data are consistent.

As ALU only take inputs from instruction (immediate numbers) and register files (data stored in registers), the inputs are always ready when the ALU needs them.

For conditional branches, this happens in ID stage, if by saying registers are read at the end of the ID stage we still allow some time for the logic gates and arithmetic units to compute, the conditional branch selection has all the inputs ready when needed. Otherwise, we need a forwarding mechanism to forward the data (that is needed by the conditional branch selector and is already computed in the EX stage but not yet written into register) to have the branch selection done before ID stage finishes.

Therefore, we need not forwarding mechanisms under the condition that we allow time for the branch selection time after reading the register needed at the end of ID stage.

Last two iterations

```
[IF][ID][EX] XOR $2, $3, $14
    [IF][ID][EX] Load $0, $13, +0 ($13 is ready as soon as EX begins)
        [IF][ID][EX] Add $1, $13, $1 ($13 is read at the end of ID)
            [IF][ID][EX] Add $0, $12, $0
                [IF][ID][EX] Add $2, $11, $2
                    [IF][ID][EX] Bne $14, -24 (Note 1)
                        [IF][xx][xx] Cpy $7, +0 is flushed (Penalty is 1)
                            [IF][ID][EX] XOR $2, $3, $14
                                [IF][ID][EX] Load $0, $13, +0
                                    [IF][ID][EX] Load $0, $13, +0
                                        [IF][ID][EX] Add $0, $12, $0
                                            [IF][ID][EX] Add $2, $11, $2
                                                [IF][ID][EX] Bne $14, -24 (Note 2)
                                                    [IF][ID][EX] Cpy $7, +0 (Note
3)
```

- (Note 1) Branch taken, flush one instruction that is in IF, nothing happens in EX
- (Note 2) Branch not taken
- (Note 3) A useless instruction

Placeholder.