Report on CAN-PR1

Submitted by Cyril WU, 5132609029, Hugo BI, 5132609036, Nicolas YING, 5030369019.

1 Two's Complement arithmetic

Wir laden Sie ein, den vollständig kommentierten Code zu überprüfen.

2 MIPS Instruction Set

We translate the binary code to assembly code ar first:
0: 000000 00100 00000 00011 00000 100101
00 04(4) 00(0) 03(3) 00 25
or $v1, $a0, $zero

4 : 001001 00100 00100 0000000000101000
09 04(4) 04(4) 0028
addiu $a0, $a0, 40

8 : 000000 00000 00000 00010 00000 100101
00 00 00 02(2) 00 25
or $v0, $zero, $zero

C : 100011 00011 00101 0000000000000000
23 03(3) 05(5) 0000
lw $a1, $v1, 00

10: 001001 00011 00011 0000000000000100
09 03(3) 03(3) 0004
addiu $v1, $v1, 4

14: 000101 00011 00100 1111111111111101
05 03(3) 04(4) fffd
bne $a0, $v1, 65533

18: 000000 00010 00101 00010 00000 100001
00 02 05 02 00 21
addu $v0, $v0, $a1

1C: 000000 11111 00000 00000 00000 001000
00 1f 00 00 00 08
jr $ra

20: 000000 00000 00010 00010 00000 100011
00 00 02(2) 02(2) 00 23
subu $v0, 00, $v0

The assmbly code is as follow

```
0: or $v1, $a0, $zero
4: addiu $a0, $a0, 40
8: or $v0, $zero, $zero
C: lw $a1, $v1, 00
10: addiu $v1, $v1, 4
14: bne $a0, $v1, 65533
18: addu $v0, $v0, $a1
```

```
1C: jr $ra
20: subu $v0, 00, $v0
```

The 0x14 instruction implements a branch jump. It can be translated as follow: If $a0 is not equal to $v1, the processor jumps to the introduction 65533 * 4 + 4 + PC. 65533 can be seen as a negative number -3, because this number is 16 bits long. Thus, the processor will jump to 0x0C. If $a0 is equal to $v1, the processor will not jump. This process corresponds to a do-while loop.
In addition, the return value is hold in $v0.

A more C like code

```
void func(int *a0){//may be array a[]
    int *v1 = a0; //one int is 4 bytes
    a0 = a0 + 10;
    int v0 = 0;
    do{
        a1 = *v1;
        v1++;
    }while(a0 != v1);
    v0 = v0 + a1;//v0 = a[9]
    return;
    v0 = -v0;//return -v0
}
```

This function recive a int array a and return -a[9]

3 Processor Design

3.1 Instruction Set Structure

Instruction Definition

Imporant note: All numbers are signed

```
[00][01][02][03][04][05][06][07][08][09][10][11][12][13][14][15]
▄▄▄▄____________________________________________▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄ : type 1
▄▄▄▄____▄▄▄▄____________________▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄________________ : type 2
▄▄▄▄____▄▄▄▄▄▄▄▄________________▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄________________ : type 3
▄▄▄▄____▄▄▄▄▄▄▄▄________________________________▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄ : type 4
```

Different thickness represents separate bit usage space.

Type 1 for conditional branch (bne in assembly)

```
▄▄1▄____immediate_number________________________▄▄register▄addr▄
[00][01][02][03][04][05][06][07][08][09][10][11][12][13][14][15]
[00] Fixed 1, similar to opcode
[01 - 11] signed immediate number
[12 - 15] register to be compared with 0
```

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.

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

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

Type 2 for load and store

```
▄▄0▄__0_▄▄▄▄_immediate_operand__▄reg▄read▄add▄2▄_reg_r/w__add_1_
[00][01][02][03][04][05][06][07][08][09][10][11][12][13][14][15]
[00] Fixed 0, similar to opcode
[01] Fixed 0, opcode extension
[02] funct bit
[03 – 07] signed immediate number, offset in memory address
[08 – 11] register address containing memory address
[12 – 15] register containing the value / to be written with the memory content
```

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.

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

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

- Let's consider the first case, where numbers are copied on conserving the negativity of the immediate numbers. That's to say, if a negative number (e.g. 11111111) is being copied, all the more significant bits are completed with 1.

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.

- The second case, where copy is only copying the bits (thus the signed property is neglected), the problem becomes much simpler.

1. Cpy $1, −1 ($1 = +255)

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

2. Cpy $2, +8 ($2 = +8)

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

3. Sll $1, $2, $3 ($3 = 65280)

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

4. Cpy $1 −1 ($1=65535) (while the higher bits are not impacted.)

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

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

Please refer to the last page.

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.

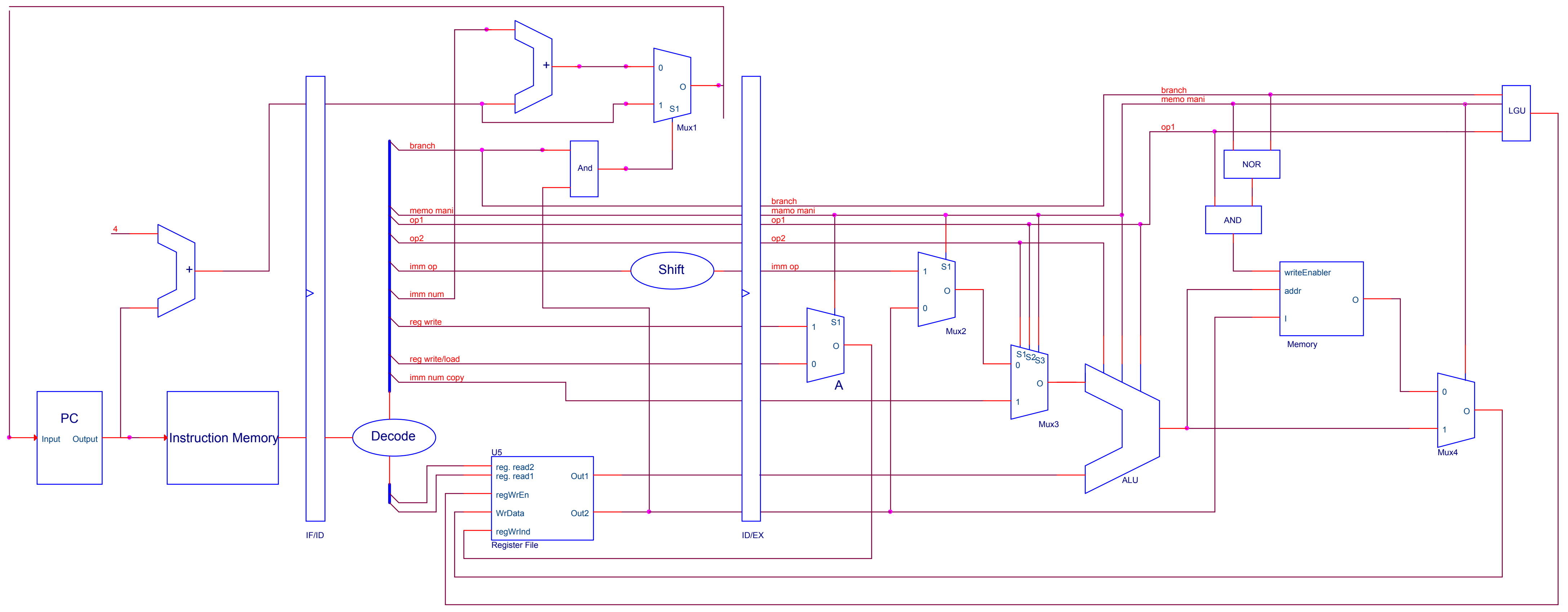
Last two iterations

```
...
[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

In conclusion, with the assumption that we do not have structural hazards, the 9 first iterations each has 1 instruction being flushed, the last iteration has one instruction that is designed to do nothing (to prevent the exhaustion of instructions).

So the total penalty is 10, 9 if not counting the last Cpy instruction which does nothing.



Decode bit spaces:

conditional branch: [00], type J, I, R, set to 1 if it's a bne instruction
memo mani(pulation): [01], type I, R, set to 0 if it reads/writes memory
op1: [02], type I, R, 0/1 meaning r/w to the memory in type I
op2: [03], type R
imm_op(erand): [03-07], type I, memory address offset
imm_num(ber): [02-11], type J, signed immediate number to be copied
reg_write(_index): [04-07], type R
reg_write_load: [12-15], type I, register being written in memory manipulation instructions
imm_num_copy: [04-11], type R' (type 4 in our taxonomy), signed number being copied
reg_read2: [08-11], type R, I
reg_read1: [12-15], type R, and type I if the memory is being written with a value from this regist er.

And for PC: Only if Out2 isn't 0 and branch bit is active, the branch would be taken in Mux1
NOR for Memory: only branch isn't active and memo_mani is 0, it is a type I instruction.
AND for Memory: only if it is a type I instruction and op1 is 1 (writing to mem) the writing is enabled.
LGU: output 1 to enable register writing for branch inactive & ((memo active & op1 = 0) or memo inactive), otherwise, disable register writer in case the output is not what we desired.

Multiplexers:
A: if memo_mani is 1, it's a load/store instruction, reg_write won't be used.
Mux2: if memo_mani is 0, we pass shifted imm_op to ALU to be added with reg_read_1

Mux3: if memo_mani is 0, we pass shifted imm_op (input 0), if memo_mani is 1, input 0 is reg_read_2.
If memo_mani is 1 and op1 is 0, pass reg_read_2,
if memo_mani is 1 and op1 is 1 and op2 is 0, pass reg_read_2,
otherwise, pass imm_num_copy.

ALU: if memo_mani is 1, do addition,
otherwise, if op1 is 0, op2 is 0, do addition,
op1 is 0, op2 is 1 do left shift, op1 is 1, op2 is 0 do XOR,
op1 is 1, op2 is 1, do copy (the inputs are right data with mux2 and mux3)

Mux4: if memo_mani is 0, the data to be written (or not) is from memory, otherwise it is computed by ALU.