Report of Assignment 3

Ang, Chen
118010009
The Chinese University of Hong Kong, Shenzhen

Introduction

In this project, we designed an ALU (Arithmetic Logical Unit) component (with its control unit and others) in a MIPS-like processor using Verilog HDL. The ALU of a processor, as the name suggests, is an integrated circuit specialized in performing arithmetic and bit wise logical operations on (integer) binary numbers. The simplest abstraction of an ALU is represented as the following. It takes in three input signals, A, B, and OP, with A and B being the two operands for the operation specified by the operation code OP, conducts that operation via some integrated circuit in it, and spits out a result Y. In many architectures (and for this project), the ALU also has an additional1-bit output signal called status, which encodes supplemental information about the result of the operation just performed. Typical status signals include carry-out, zero, negative, overflow, parity, etc. The status outputs are often stored as a whole in one multi-bit register called the status register. The zero signal is useful in deciding conditional branching in this project.

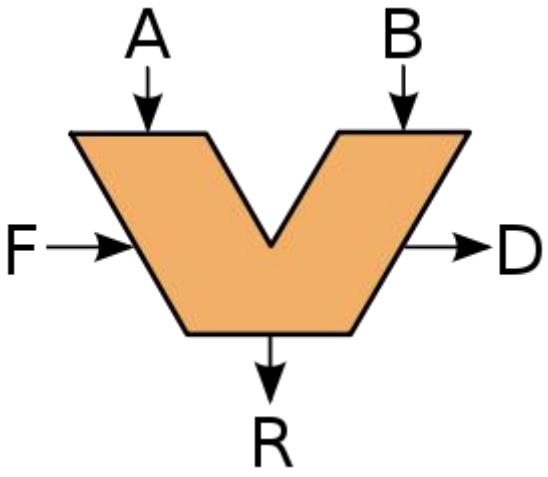

Fig. 1. *An abstraction of ALU. A and B are the operands; R is the result; F(OP) decodes the operation; D is the output status*

What our ALU Does

The ALU implemented in this project is slightly more complicated than introduced above in that it also contains some part of the control unit, sign extension, a PC and the circuit for incrementing it (although PC cannot be used for fetching instructions yet; it must be done manually.) For this reason it might be more appropriate to call our “ALU” a watered-down version of a CPU. However, the crucial incompleteness if we were to call it a CPU are that a) it has no interaction with the memory, be it data or instruction field. Instead, instructions are passed in manually to the “ALU” and no save/load instructions can actually be performed; and b) it does not contain a register file. Registers are directly inputted rather than fetched by the register number in the instruction.

Three 32-bit binaries are passed in as inputs of our “ALU”. There are two general registers, `rs` and `rt`; and a 32-bit MIPS instruction code to specify the instruction `op/funct` codes and provide `shamt` and `imm` values. The control unit then processes the opcode and function codes and tell the ALU which operation to perform through `ALUctrl`. Other control signals are outputted as well, including

- `srcActrl(0~1)/srcBctrl(0~4)`, which determines the source of operand A/B;
- `branchEQ & branchNE`, which, combined with `ZeroFlag`, control loading branch target into PC.

The following four control signals are not actually controlling data read/write in this particular project but are crucial for the next one (project 4.)

- `memToReg`, which determines data source to be written into the register;
- `memWrite/regWrite`, which enables data write into memory/register;
- `regDst`, which determines the destination register to write data into.

The main ALU conducts the operation specified by `ALUctrl` on operands `SrcA` and `SrcB` and output the result. Other components of the circuit such as `PCSrc` are also determined by the output control signals to do their job (e.g. deciding the source of PC.)

The instruction set supported by the “ALU” is a subset of the MIPS instruction set and will be introduced in next section.

Implementation Details

Block Diagram

The major components of the ALU are shown in the following block diagram.

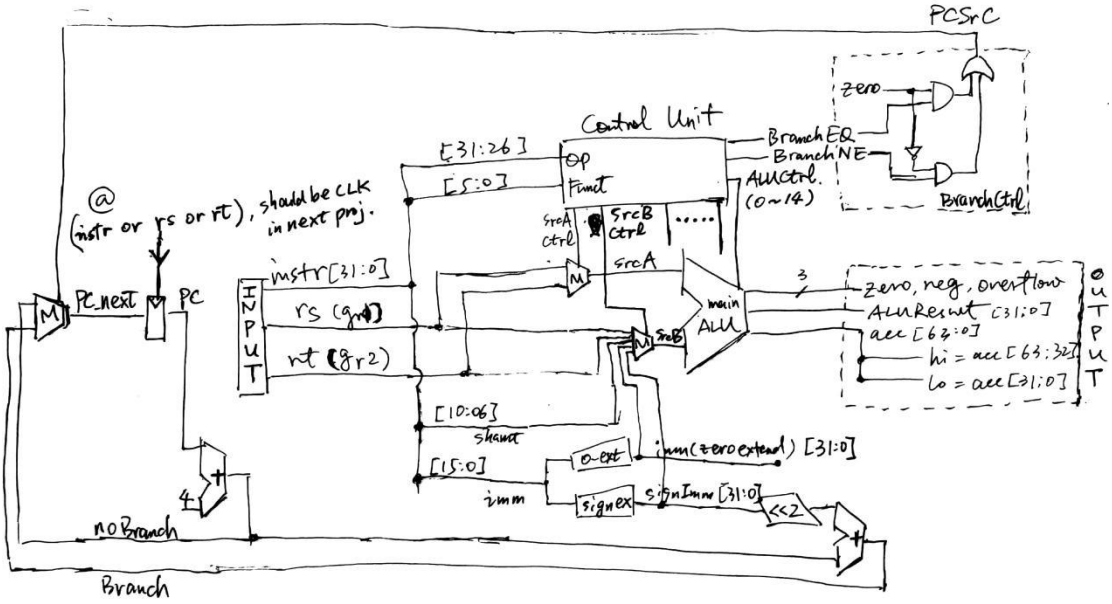

Fig. 2. *Block diagram of the ALU and relevant components (control signals that are not used in this project are omitted)*

Control Unit

The control unit needs to generate correct control signals based upon the input `op/funct` codes: the `ALUctrl` ranging from 0 to 14 that encodes the operation used in the instruction; `srcActrl/srcBctrl/regDst` encoding the sources and destination of data to the ALU. This instruction-based encoding is realized through Verilog's case instruction on the `op/funct` codes segmented from the instruction. The specific encoding scheme used in the project is listed in a table below.

Instr	*Operation*	*ALUctrl*	*srcActrl*	*srcBctrl*	*regDst*
add	+	0	rs (1)	rt (0)	rd (1)
addu					
addi				signImm (3)	rt (0)
addiu					
lw					
sw					

sub	-	1		rt (0)	rd (1)
subu					
beq					x
bne					
mult	× (±)	2			rd (1)
multu	× (∅)	3			
div	÷ (±)	4			
divu	÷ (∅)	5			
and	&	6			
andi				imm (2)	rt (0)
nor	~|	7		rt (0)	rd (1)
or	|	8			
ori				imm (2)	rt (0)
xor	^	9		rt (0)	rd (1)
xori				imm (2)	rt (0)
slt	slt (±)	10		rt (0)	rd (1)
slti					
sltu	slt (∅)	11			
sltiu					
sll	<<	12	rt (0)	shamt (4)	
sllv				rs (1)	
srl	>>	13		shamt (4)	
srlv				rs (1)	
sra	>>>	14		shamt (4)	
srav				rs (1)	

±: signed; ∅: unsigned

Table 1. *Grouping of supported instructions by operations(ALUctrl.) srcA, srcB, regDst and their encoding listed.*

Main ALU

The main ALU conducts the operation encodes by `ALUctrl` on two input operands. Thus ALU needs to first decode the `ALUctrl`. This decoding is again realized using the case instruction. What remains is to simply to tell the ALU to conduct the basic operations listed above as most of these binary operations are built into the Verilog. Some operations among them do need to be distinguished between signed and unsigned versions, while ***regular operators in Verilog are by default unsigned*** . Luckily the Verilog has an built-in function `$signed()` which handles this scenario for us.

By directly converting both operands `a, b` into `$signed(a)` and `$signed(b),` any operation `X` on them will automatically become signed as well. Another type of operation worth mentioning is the shifting. Sometimes when the shift amount is indicated by a register as in `sllv` or `srav,` the amount for shifting could be too wide (>32) and cause an error. This is handled by reducing the shift amount modulo 32, or equivalently, adding it with `0x1F,` which prevents undefined shifting behavior.

Sign and Zero Extensions

Sign extension is realized by the built-in `$signed()` function in Verilog.
Zero extension needs no explicit statements. It is done automatically in Verilog.

Instruction Examples

Sample outputs are listed for all supported instructions below with input registers set at:

```
gr1 <= 32'b1000_1001_1001_1001_1001_1001_1001_1001;
gr2 <= 32'b0101_1101_1101_1101_1101_1101_1101_1101;
```

The outputs are in the following format (in hex):

```
instruction:op:func:gr1:gr2:srcA:srcB:out:hi:lo:z:o:n:PCSrc:memW:regW:regD:mem2Reg:ALUctrl:PC
```

`z`: Zero; `o`: Overflow; `n`: Negative; `memW`: Memory Write; `regW`: Register Write; `regD`: Register Destination

- **add**

```
014b4820:00:20:89999999:5ddddddd:89999999:5ddddddd:e7777776:xxxxxxxx:xxxxxxxx:0:0:1:0:0:0:1:0:0:00000000
```

- **addu**

```
02328021:00:21:89999999:5ddddddd:89999999:5ddddddd:e7777776:xxxxxxxx:xxxxxxxx:0:0:1:0:0:0:1:0:0:00000004
```

- **sub**

```
014b4822:00:22:89999999:5ddddddd:89999999:5ddddddd:2bbbbbbc:xxxxxxxx:xxxxxxxx:0:1:0:0:0:0:1:0:1:00000008
```

- **subu**

```
02328023:00:23:89999999:5ddddddd:89999999:5ddddddd:2bbbbbbc:xxxxxxxx:xxxxxxxx:0:1:0:0:0:0:1:0:1:0000000c
```

- **beq**

```
112a0001:04:01:89999999:5ddddddd:89999999:5ddddddd:2bbbbbbc:xxxxxxxx:xxxxxxxx:0:1:0:0:0:0:1:0:1:00000010
```

- **and**

```
014b4824:00:24:89999999:5ddddddd:89999999:5ddddddd:09999999:xxxxxxxx:xxxxxxxx:0:1:0:0:0:0:1:0:6:00000014
```

- **or**

```
02328025:00:25:89999999:5ddddddd:89999999:5ddddddd:dddddddd:xxxxxxxx:xxxxxxxx:0:1:1:0:0:0:1:0:8:00000018
```

- **nor**

```
02328027:00:27:89999999:5ddddddd:89999999:5ddddddd:22222222:xxxxxxxx:xxxxxxxx:0:1:0:0:0:0:1:0:7:0000001c
```

- **xor**

```
012a4026:00:26:89999999:5ddddddd:89999999:5ddddddd:d4444444:xxxxxxxx:xxxxxxxx:0:1:1:0:0:0:1:0:9:00000020
```

- **div**

```
012a001a:00:1a:89999999:5ddddddd:89999999:5ddddddd:d4444444:e7777776:ffffffff:0:1:1:0:0:0:1:0:4:00000024
```

- **divu**

```
012a001b:00:1b:89999999:5ddddddd:89999999:5ddddddd:d4444444:2bbbbbbc:00000001:0:1:1:0:0:0:1:0:5:00000028
```

- **mult**

```
012a0018:00:18:89999999:5ddddddd:89999999:5ddddddd:d4444444:d4962fc9:9147ae15:0:1:1:0:0:0:1:0:2:0000002c
```

- **multu**

```
012a0019:00:19:89999999:5ddddddd:89999999:5ddddddd:d4444444:32740da6:9147ae15:0:1:1:0:0:0:1:0:3:00000030
```

- **sll**

```
00011040:00:00:89999999:5ddddddd:5ddddddd:00000001:bbbbbbba:xxxxxxxx:xxxxxxxx:0:1:1:0:0:0:1:0:c:00000034
```

- **srl**

```
00118282:00:02:89999999:5ddddddd:5ddddddd:0000000a:00177777:xxxxxxxx:xxxxxxxx:0:1:0:0:0:0:1:0:d:00000038
```

- **srlv**

```
02718006:00:06:89999999:5ddddddd:5ddddddd:89999999:0000002e:xxxxxxxx:xxxxxxxx:0:1:0:0:0:0:1:0:d:0000003c
```

- **sra**

```
00118283:00:03:89999999:5ddddddd:5ddddddd:0000000a:00177777:xxxxxxxx:xxxxxxxx:0:1:0:0:0:0:1:0:e:00000040
```

- **srav**

```
02718007:00:07:89999999:5ddddddd:5ddddddd:89999999:0000002e:xxxxxxxx:xxxxxxxx:0:1:0:0:0:0:1:0:e:00000044
```

- **slt**

```
0233802a:00:2a:89999999:5ddddddd:89999999:5ddddddd:00000001:xxxxxxxx:xxxxxxxx:0:1:0:0:0:0:1:0:a:00000048
```

- **sltu**

```
0233802b:00:2b:89999999:5ddddddd:89999999:5ddddddd:00000000:xxxxxxxx:xxxxxxxx:1:1:0:0:0:0:1:0:b:0000004c
```

- **addi (imm = 0x8000)**

```
21498000:08:00:89999999:5ddddddd:89999999:ffff8000:89991999:xxxxxxxx:xxxxxxxx:0:0:1:0:0:0:0:0:0:00000050
```

- **addiu (imm = 10)**

```
26300064:09:24:89999999:5ddddddd:89999999:00000064:899999fd:xxxxxxxx:xxxxxxxx:0:0:1:0:0:0:0:0:0:00000054
```

- **andi (imm = 10)**

```
32300064:0c:24:89999999:5ddddddd:89999999:00000064:00000000:xxxxxxxx:xxxxxxxx:1:0:0:0:0:0:0:0:6:00000058
```

- **ori (imm = 10)**

```
36300064:0d:24:89999999:5ddddddd:89999999:00000064:899999fd:xxxxxxxx:xxxxxxxx:0:0:1:0:0:0:0:0:8:0000005c
```

- **xori (imm = 10)**

```
3a300064:0e:24:89999999:5ddddddd:89999999:00000064:899999fd:xxxxxxxx:xxxxxxxx:0:0:1:0:0:0:0:0:9:00000060
```

- **slti (imm = 10)**

```
2a300064:0a:24:89999999:5ddddddd:89999999:00000064:00000001:xxxxxxxx:xxxxxxxx:0:0:0:0:0:0:0:0:a:00000064
```

- **sltiu (imm = 10)**

```
2e300064:0b:24:89999999:5ddddddd:89999999:00000064:00000000:xxxxxxxx:xxxxxxxx:1:0:0:0:0:0:0:0:b:00000068
```

- **beq (imm = -1)**

```
1211ffff:04:3f:89999999:5ddddddd:89999999:5ddddddd:2bbbbbbc:xxxxxxxx:xxxxxxxx:0:1:0:0:0:0:0:0:1:0000006c
```

- **bne (imm = -1)**

```
1611ffff:05:3f:89999999:5ddddddd:89999999:5ddddddd:2bbbbbbc:xxxxxxxx:xxxxxxxx:0:1:0:1:0:0:0:0:1:00000070
```

- **lw (imm = 100)**

```
8e300064:23:24:89999999:5ddddddd:89999999:00000064:899999fd:xxxxxxxx:xxxxxxxx:0:0:1:0:0:0:0:1:0:00000070
```

- **sw (imm = 100)**

```
ae300064:2b:24:89999999:5ddddddd:89999999:00000064:899999fd:xxxxxxxx:xxxxxxxx:0:0:1:0:1:1:0:0:0:00000074
```

- **sllv**

```
01494004:00:04:89999999:5ddddddd:5ddddddd:89999999:ba000000:xxxxxxxx:xxxxxxxx:0:0:1:0:0:0:1:0:c:00000078
```

PC and Branching

`PC` is updated to `PCnext` each time `@ rs or rt or instr`. (to be changed to `@ clk` in the next project) The source of `PCnext` is decided via the control signal

```
PCSrc = (branchEQ & zero) | (branch & !zero)
```

to be either

```
PC + 4
```

if `PCSrc = 0` (no branch), or

```
PC + 4 + (signImm << 2)
```

if `PCSrc = 1` (branch).