Documentation of CPU project

Mahdi Haghverdi
Seyed Hussein Husseini

June 12, 2023

Contents

Chapter 1

Assembly

In this simple and short chapter, we will briefly explain the assembly language of our CPU

The assembly language of our CPU looks like this:

```
add $a1, $zero, $a2
```

But how are is this language is formatted for us? look at this table:

Instruction format	command	register	register	register	don't care	funct
R-type	add	$a1	$a2	$a3	000...000	001

Table 1.1: R-type instruction assembly

Which means: add $a1 to $a2 and store the value in $a3 registers.

Instruction format	command	register	register	immediate value
V-type	andi	$l1	$t8	00101...01002

Table 1.2: V-type instruction assembly

Which means: and $l1 with the imm value and store the result in $t8.

Chapter 2

ISA

In this chapter we will introduce the ISA of our CPU. Many different and important aspects of the ISA will be covered. Covered topics are:

- ISA Table
- ISA Instruction Formats
- ISA Encoding

2.1 ISA Table

The given ISA to our group consists of 12 commands, spread into arithmetic, logical, branching and memory categories.

Table 2.1: ISA table

Type	Assembly Instruction
Arithmetic	`add, sub, mul`
Logical	`and, or, nor, nand, andi, ori`
Branching	`bnq`
Memory	`lw, sw`

Do note that in our CPU all ISA instructions mentioned in table 2.1, deal with registers to get and store data; no operation can be dealt with direct access to memory.

2.2 ISA Instruction Formats

In this CPU we have two types of instruction formats: 1. `R-type`[1] 2. `V-type`[2]; and in this section we will go through details of this formats.

These are the the divided assembly instructions of the CPU:

Table 2.2: `r-type` and `v-type` instructions

Type	Assembly Instruction	Count
`R-type`	`add, sub, mul, and, or, nor, nand`	7
`V-type`	`bnq, lw, sw, andi, ori`	5

Our CPU instructions are 32-bit long; the instruction format for **recognizing the instruction**[3], looks like this:

Table 2.3: Instruction format of `r-type` and `v-type` instructions

Type	Opcode	Rs	Rt	Rd	Don't Care	Funct
`R`	`000`	`nnnn`	`nnnn`	`nnnn`	`14-bit`	`nnn`
Type	Opcode	Rs	Rt	immediate value		
`V`	`nnn`	`nnnn`	`nnnn`	`21-bit`		

[1] Register format instruction
[2] Value format instruction
[3] encoding the assembly instruction in binary

2.3 ISA Encoding

All CPUs only understand zeros and ones, in order to tell the CPU to do this and to do that, we have to encode the instructions.

The instruction code table is below:

Table 2.4: ISA encoding

Instruction (`r-type`)	Opcode	Funct
`add`	`000`	`001`
`sub`	`000`	`010`
`mul`	`000`	`011`
`and`	`000`	`100`
`or`	`000`	`101`
`nor`	`000`	`110`
`nand`	`000`	`111`

Instruction (`v-type`)	Opcode
`bnq`	`111`
`lw`	`001`
`sw`	`010`
`andi`	`100`
`ori`	`101`

Chapter 3

Registers

In this chapter we will look at the register file of the CPU and the design decisions of this important part. Covered topics are:

- Number Of Registers
- Naming Conventions of Registers
- Capacity Of Registers
- Register file design

All the operations which take place in the CPU get their operands and store their results in the register file. For example the instruction:

```
add $r1, $r2, $r3
```

Which means `$r3 = $r1 + $r2`, the data are fetched from the registers `$r1`, `$r2` and `$r3`.

There are some decisions for the design of the register file of the CPU which will be discussed below.

3.1 Number Of Registers

After discussions between the team members we decided to put 16 registers in the register file of the CPU.

With this number of registers, we can access them by a four-bit number, because $16 = 2^4$

3.2 Naming Conventions of Registers

The registers are names as follows:

1. `$zero`
2. `$at`
3. `$a#`
    - `$a1`
    - `$a2`
    - `$a3`
4. `$l#`
    - `$l1`
    - `$l2`
    - `$l3`
5. `$t#`
    - `$t1`
    - `$t2`
    - `$t3`
    - `$t4`
    - `$t5`
    - `$t6`
    - `$t7`
    - `$t8`

3.2.1 Register Specifications

These register specifications are done just for convenience and safety of the compiled programs. For example, a logical operation operands are better to be placed in `$l#` registers. This convention will guarantee the data safety during the program execution.

3.2.1.1 `$zero` register

Because the value `0` is used a lot, we put it here. This register will not be overwritten.

3.2.1.2 `$at` register

This is the register provided for the assembler to use on its own.

3.2.1.3 `a#` registers

`a#` registers are named after the word `arithmetic`. These registers are specific for arithmetic operations.

3.2.1.4 `l#` registers

`l#` registers are named after the word `logical`. These registers are specific for arithmetic operations.

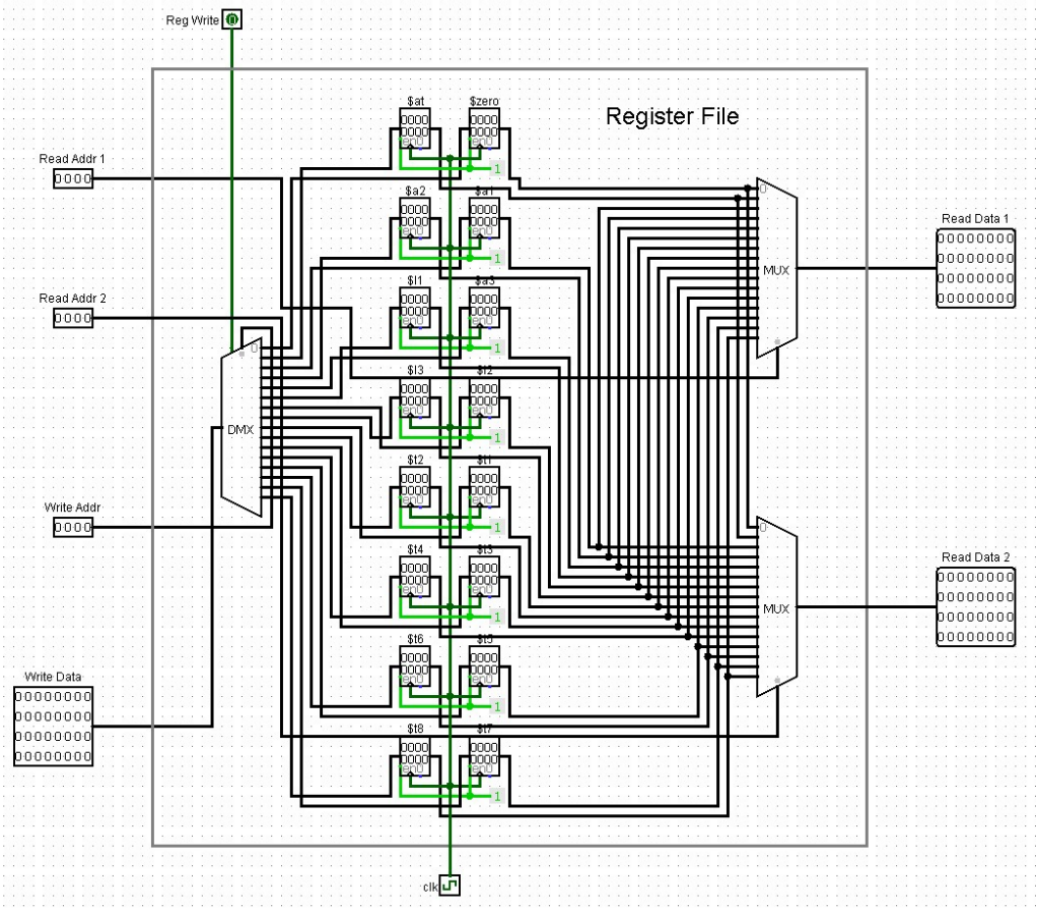
3.2.1.5 `t#` registers

`t#` registers are named after the word `temporary`. These registers can be used freely by the compiler or assembly programmer.

3.3 Capacity Of Registers

This CPU's registers are 32-bit registers with the capability of parallel load and read.

3.4 Register file design



Chapter 4

Control Unit

Control Unit and Data Path are the two hearts of all CPUs. In this chapter we will show the design of our CPU's control unit.

Control unit, produces some signals to power on or power off some parts of the data path to control the data flow between the gates, registers, the alu and multiplexers of the data path. The table below shows all the control unit signals we need for our ISA (discussed in ISA chapter) Here is the table of all

4.1 Control Unit Signals Table

The meaning of zeros and ones in the table, are explained in the footnote of the table.

Table 4.1: Control Unit Signals

Control Signal	add	sub	mul	and	or	nor	nand	bnq	lw	sw	andi	ori
RegDst	1[a]	1	1	1	1	1	1	x	0[b]	x	0	0
ALUSrc	0[c]	0	0	0	0	0	0	0	1[d]	1	1	1
MemToReg	0[e]	0	0	0	0	0	0	x	1[f]	x	0	0
RegWrite	1[g]	1	1	1	1	1	1	x	1	x	1	1
MemWrite	0	0	0	0	0	0	0	0	0	1[h]	0	0
Branch	0	0	0	0	0	0	0	1[i]	0	0	0	0
ALUctrl	add	sub	mul	and	or	nor	nand	sub	add	add	and	or

[a] Rd
[b] Rt
[c] a register
[d] immediate value
[e] from ALU result to a register
[f] from memory to a register
[g] write to register from ALU
[h] write to memory from ALU
[i] put one in the branch's **and** gate.

4.2 Control Unit Design

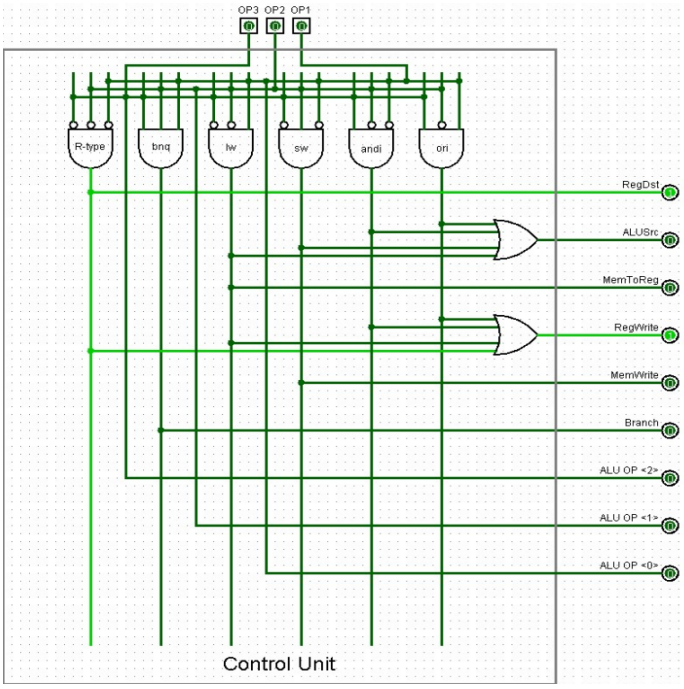


Figure 4.1: Main Control Unit Design

Chapter 5

ALU

ALU is the one of the main units in all CPUs. It is the Arithmetic and Logic Unit, which actually executes the instruction. In this chapter we will look at the ALU of our CPU. Covered topics are:

1. ALU Operations
2. ALU Encoding
3. ALU Design
4. ALU Main Control Unit

5.1 ALU Operations

Our ALU according to the given ISA, has 7 different operations:

1. `add (a + b)`
2. `sub (a - b)`
3. `mul (a * b)`
4. `and (a & b)`[1]
5. `or (a | b)`[1]
6. `nor (∼(a | b))`[1]
7. `nand (∼(a & b))`[1]

5.2 ALU Encoding

CPUs decide what to do according to the instruction, instructions are coded in a special way for each CPU; however ALUs do follow this behaviour. Here we see the ALU encoding:

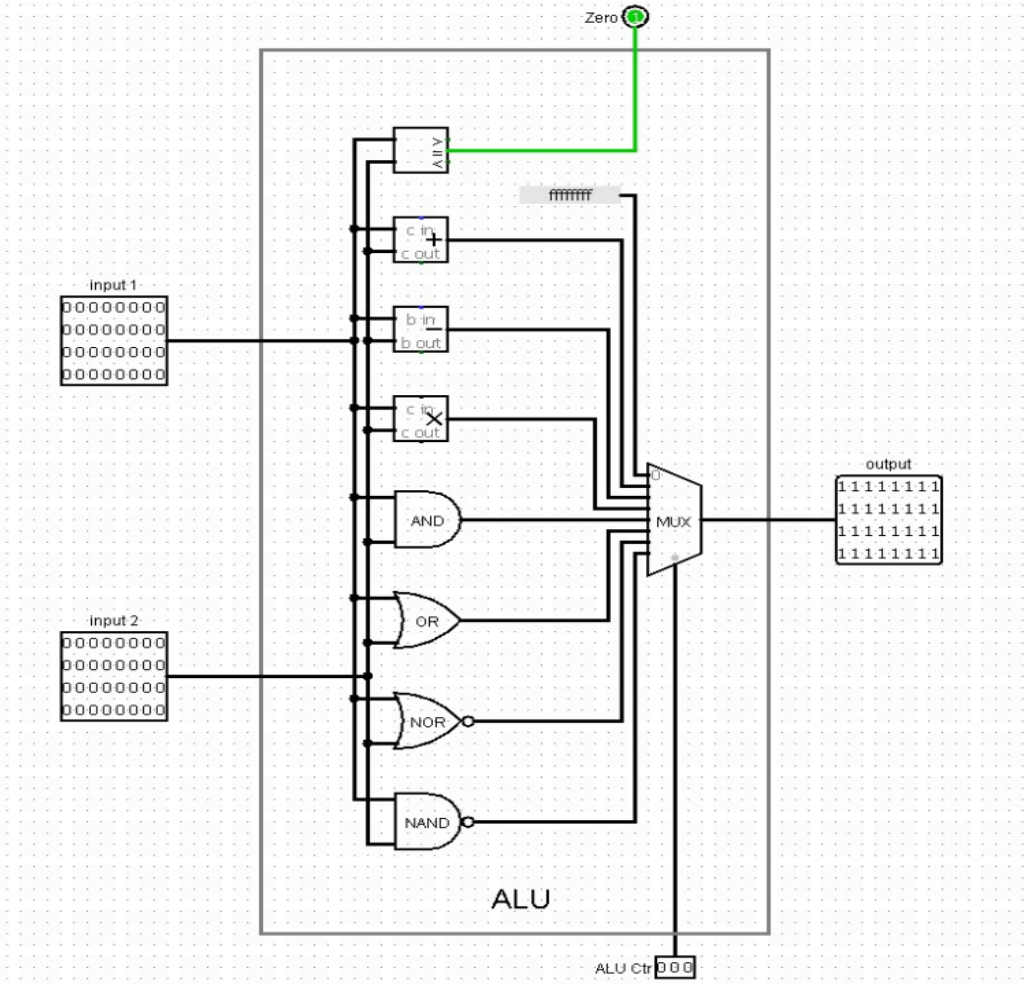
op	code
add	001
sub	010
mul	011
and	100
or	101
nor	110
nand	111

op	funct	alu-code
add	001	001
sub	010	010
mul	011	011
and	100	100
or	101	101
nor	110	110
nand	11	111

op	opcode	alu-code	
bnq	111	sub	010
lw	001	add	001
sw	010	add	001
andi	100	and	100
ori	101	or	101

[1] These logical operations are done bit by bit

5.3 ALU Design

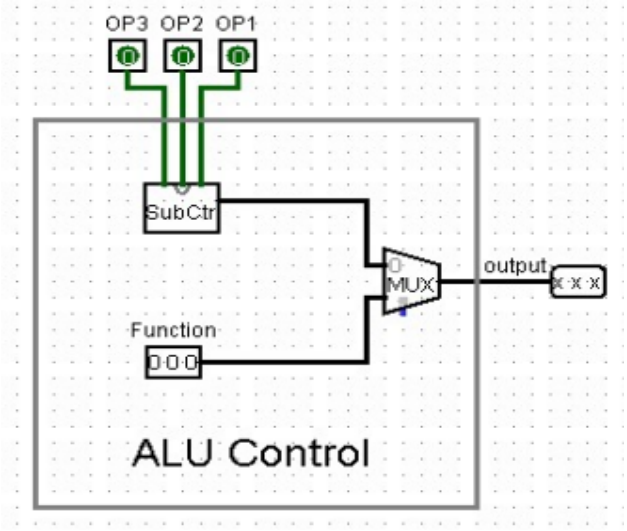


5.4 ALU Main Control Unit

All ALUs has a select port which tells them what to do! But how this select signal is produced? Well, according to the instructions which is fetched, this signal is produced. A hardware section in our CPU does that which is called: `ALU Sub-Control Unit`.

This unit gets the `opcode` (and `funct`) from the instruction and decides what to do.

It is very simple and here it is the unit:



5.4.1 ALU Sub-Control Unit

There is a small unit inside this unit, which again decides what to produce according to the `opcode` of the instruction: