Zhixun Tan / 谈至勋

2012011120

EE25

Tsinghua University

Digital Logic and CPU Report

MIPS ALU, Assembler, Single-cycle CPU

The MIPS ALU

ALU stands for arithmetic/logic unit, which is built with simple combinational logic circuits.

In order to support the subset of MIPS assembly, the ALU needs to support the following operations:

ADD, SUB, AND, OR, XOR, NOR, A, SLL, SRL, SRA, EQ, NEQ, LT, LEZ, GEZ, GTZ

The ALU takes in a control signal `ALUFun[5:0]` to specify which operation to perform.

The ADD/SUB operation is the same for both signed and unsigned numbers, since we are using the two's compliment data representation. But, to determine whether there is an overflow, the two cases need to be treated differently.

Overflow detection for signed operations:

If we are adding two signed numbers, the first thing to notice that adding a non-negative number and a negative number never triggers an overflow. Suppose the range of our integer representation is [-M, M – 1], then [-M, -1] + [0, M – 1] = [-M, M – 2], and we are in the safe range.

The problem comes when we are adding two non-negative numbers or two negative-numbers. [0, M – 1] + [0, M – 1] = [0, 2M – 2] which is out of bound. But notice that any number in [M, 2M – 2] will be trunked and interpreted as [-M, -2], so as we are adding two non-negative numbers, we seem to get a negative answer. That is the signal of an overflow.

Similarly, if we are adding two negative numbers and we get a non-negative answer, this is also an overflow. If we are subtracting a negative number from a non-negative number, or a non-negative number from a negative number, and not getting the expected sign, it's an overflow.

Overflow detection for unsigned operations:

This time, instead of dividing numbers into non-negative and negative ones, we divide them into small and large ones. Any integers whose MSB is 1 is considered a large number, and those whose MSB is 0 is a small number. The ranges of these two kinds of integers are $[0, N-1]$ and $[N, 2N-1]$.

Only adding two small numbers are considered safe. Adding two large numbers will definitely cause an overflow because $[N, 2N-1] + [N, 2N-1] = [2N, 4N-2]$. As for adding a small number and a large number, we need to think carefully.

$[0, N-1] + [N, 2N-1] = [N, 3N-2] = [N, 2N-1] \cup [2N, 3N-2]$. Note that the portion $[2N, 3N-2]$ will be interpreted as $[0, N-2]$ so if we seem to get a smaller number as the answer, it's an overflow.

Subtracting a small number from a big number is safe, and subtracting a big number from a small number is definitely a wrong move. Subtracting a big number from a big number, or a small number from a small number must result in a small number, or it's an overflow.

For other information, such as negativity detection, please check the source code. It is well documented.

The structure of ALU

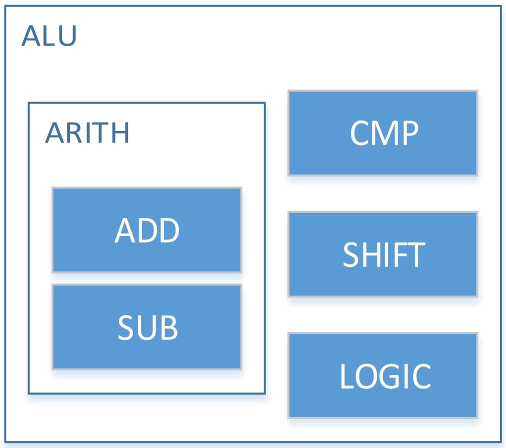


Figure 1: The structure of the ALU module

Figure 1 shows the structure of the ALU. There is a ADD and a SUB module, which are packed in a bigger ARITH module. The ARITH module takes in two operands and a control signal to specify the operation, performs the arithmetic, and generates the output as well as several signals indicating whether an overflow is triggered, whether the answer is zero, and whether it is negative.

The other three types of arithmetic are comparing, shifting, and logic operations. The CMP module takes in the output of ARITH, which gets the difference of the two operands and determine whether is greater than zero, equal to zero, or smaller than zero.

Finally, a large module ALU packs all these modules, leaving a single interface.

A piece of code: the Compare module

Here we present the actual code of the compare module.

```
module Compare(
    input wire Zero,
    input wire Overflow,
    input wire Negative,
    input wire[2:0] FT,
    output wire S
);
    parameter FT_CMP_EQ  = 3'b001;
    parameter FT_CMP_NEQ = 3'b000;
    parameter FT_CMP_LT  = 3'b010;
    parameter FT_CMP_LEZ = 3'b110;
    parameter FT_CMP_GEZ = 3'b100;
    parameter FT_CMP_GTZ = 3'b111;

    parameter ERROR_OUTPUT = 1'b1;

    assign S = (FT == FT_CMP_EQ) ? Zero : (
        (FT == FT_CMP_NEQ) ? ~Zero : (
            (FT == FT_CMP_LT) ? Negative : (
                (FT == FT_CMP_LEZ) ? (Negative | Zero) : (
                    (FT == FT_CMP_GEZ) ? (~Negative) : (
                        (FT == FT_CMP_GTZ) ? (~Negative & ~Zero) :
                            ERROR_OUTPUT
                    )
                )
            )
        )
    );

endmodule
```

As shown in the code, FT[2:0] is a control signal indicating the type of comparing we intend to do. The answer is based on some outputs of the arithmetic operation.

The MIPS Assembler

A MIPS assembler takes in MIPS assembly code, and translates it into binary machine code. Our implementation is a standard compiler approach, which divides the full translating process into several stages.

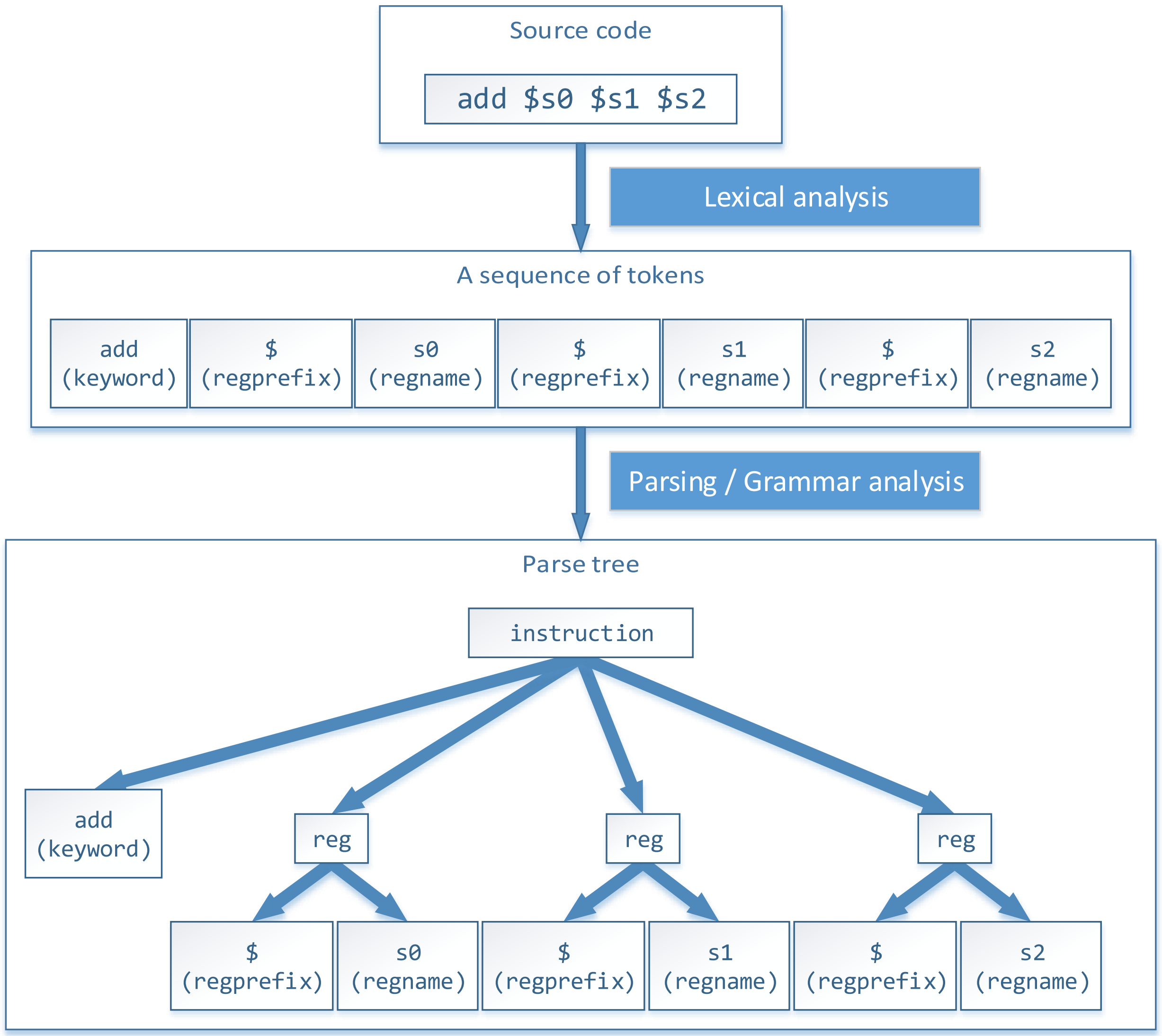


Figure 2: The full assembler front end.

First, consider the source code input. It's literally just a string. The computer cannot extract any information from this raw string unless we carefully treat it. The compiler's job is to "understand" the string by creating a suitable structure.

As shown in Figure 2, we need to organize the code in a tree structure, which is called the *parse tree*. As soon as we get the parse tree, it becomes easy to perform various kinds of operations on the code. Our assembler can then generate machine code based on this tree.

So how do we get the tree? It turns out we need two steps: lexical analysis and parsing.

Lexical analysis is the stage where we takes in the source string, and divide it into separate “words”. A “word” is called a *token* technically, so the stage of lexical analysis is also called tokenizing. Each token has a type, just like in English we have verbs, nouns, etc.

The tokens are matched with *regular expressions*. For example, a register name of the s-series can be matched through “`s[0-7]`”.

In this stage we may also ignore some characters like spaces and new-lines.

Parsing is the stage we apply a certain grammar on the tokens and thus create the tree. For example, an add instruction should contain three registers:

```
instruction : “add” reg reg reg
```

And each register should have a prefix “`$`”:

```
reg : “$” regname
```

The techniques of constructing the tree are many, but they are basically divided into two types: top-down and bottom-up. We use a well-known toolkit pair – *lex* and *yacc* to do the two stages for us. We specify the regular expressions for tokens and the grammar rules, and the toolkit performs regular expression matching and bottom-up parsing.

At the end we get the tree structure we want.

The next thing to think about is what we can do on the tree. We may modify the tree, add or delete some information to make it even more clear what the code intends to do. This is called *semantic analysis*.

Semantic analysis is the stage we finalize the tree and prepare it for the final output. For example, why do we even need the “$” in the tree, since we already know it’s a register? Which register is “s0” – it’s the 16th register. After doing all these, we are ready to generate the translated code!

Code generation is the stage we write the output. We need to traverse the tree

Testing

Now let’s see an example. Suppose we’ve written the following assembly code.

```
    .text
    addi $s0 $zero 10
loop
    add $s0 $s0 $s0
    j loop
.kernal
    jr $k0
```

Then the tokenizer would list all tokens in the code.

```
('DOT', '.')
('TEXT', 'text')
('ADDI', 'addi')
('REGPREFIX', '$')
('S', 's0')
('REGPREFIX', '$')
('ZERO', 'zero')
('NUMBER', 10)
('IDENTIFIER', 'loop')
('ADD', 'add')
('REGPREFIX', '$')
('S', 's0')
('REGPREFIX', '$')
('S', 's0')
('REGPREFIX', '$')
('S', 's0')
('J', 'j')
('IDENTIFIER', 'loop')
('DOT', '.')
('KERNAL', 'kernal')
('JR', 'jr')
('REGPREFIX', '$')
('K', 'k0')
```

Then we call the parser.

```
Parser error at or near add of line 4
```

Oops! The parser tells us that we've had an error! As we look back at our code, we find out that we've missed the ":" after the "loop" label. After correcting the code, the parser is satisfied and happily sends the tree to the code generator, which generates the ROM code for us.

```
// This ROM is automatically generated by the assembler.

module ROM(
    input [31:0] addr,
    output reg[31:0] data
);

    always @(*) begin
        case (addr)
        32'h00400000: data <= 32'h2010000A; // addi $s0 $zero 10
        32'h00400004: data <= 32'h02108020; // add $s0 $s0 $s0
        32'h00400008: data <= 32'h08100001; // j loop
        32'h80000000: data <= 32'h03400008; // jr $k0

        default: data <= 32'h00000000;
        endcase
    end

endmodule
```

The Single-cycle CPU

The single-cycle CPU runs one instruction every clock cycle. We could think of it as a finite-state machine. On every positive edge of the clock cycle, the machine changes its state. More specifically, on the posedge, these things are done:

The PC (program counter) register changes its value to the address of the next instruction;
One register might change its value;
One word of memory might be written.

Considering these operations, we must make sure that the CPU can gather enough and correct information within one clock cycle. For example, we need to know what data is going to be written to which register before the posedge where the writing is triggered.

The control unit

There are so many control signals in the CPU that the module which generates them are called the control **unit**. Basically the control unit should decide what kind of operation should the ALU do, where the data to be written comes from, and where to write the data. It is a huge module filled with combinational logics.

The memory

The memory is the storage in the computer. It is simply a huge array of bytes storing massive amounts of data. Each byte has its own address, through which we can access. However, we don’t move data one byte a time. Instead, we do this four bytes a time. For example, we want to retrieve some data at the address 0x100, what we are retrieving is the four bytes 0x100, 0x101, 0x102, and 0x103, and the four bytes form an integer.

It turns out there are two orders of the four bytes. For example, if we are storing the integer 0x102F3D42 into address 0x100 through 0x103, we can either store 0x10 in address 0x100, or store 0x42. The former one where we store the MSB in the lowest address is called big-endian, while the latter one where we store the LSB in the lowest address is called little-endian. Our implementation is a big-endian one.

The peripherals are mapped to certain addresses of memory so that controlling them is like reading and writing the memory.

Interrupt

An interrupt is triggered by an I/O device. It is a signal saying that the CPU should pause whatever it is doing and start manipulating I/O devices.

The interrupt is asynchronized. It shows up in the middle of a clock cycle. The CPU must cancel the current instruction by closing the write register and write memory signal, and set the address of the next instruction to a special I/O program. The interrupt signal is a one-bit signal given by the peripheral, and once it's triggered, it keeps being 1 until the next clock cycle. The control unit recognize this signal and set the corresponding control signals through combinational logics. The behavior of an interrupt is shown in Figure 3.

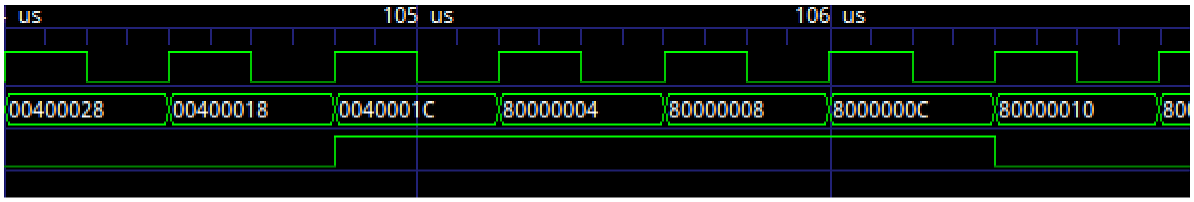
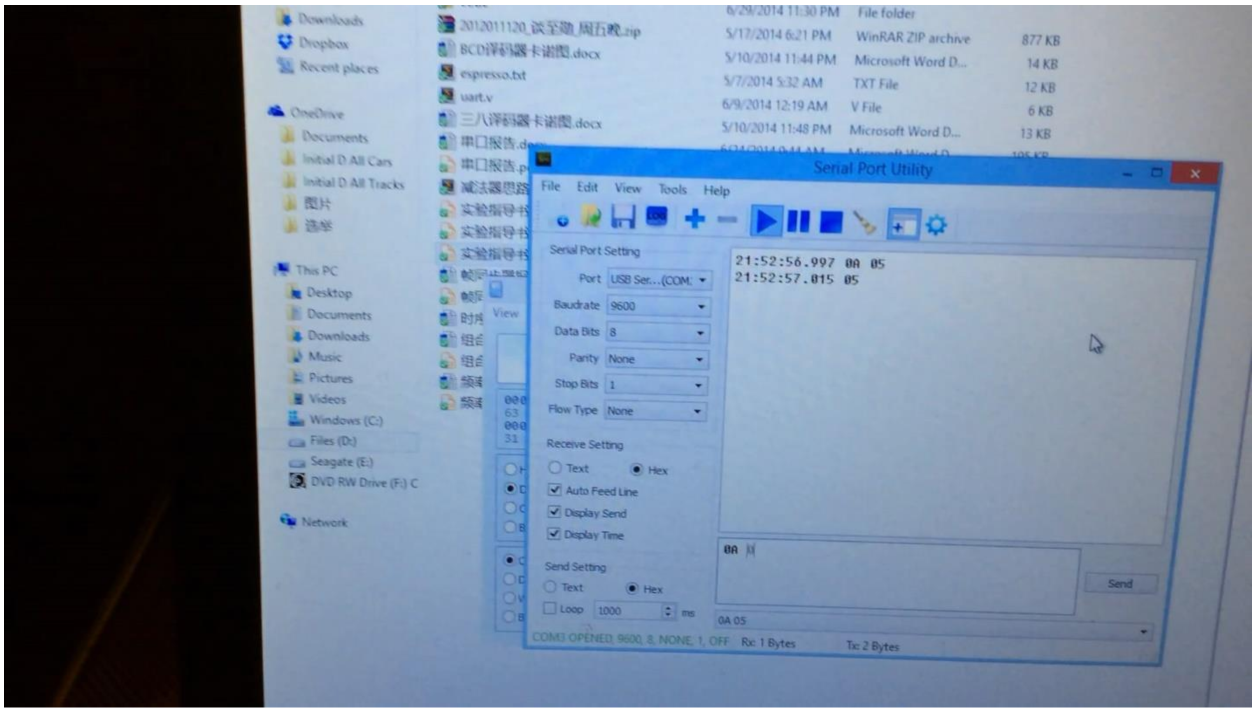


Figure 3: An interrupt is triggered and the PC is set to 0x80000004

Testing

The CPU receives the two parameters from UART, calculates the greatest common divider of them, and returns the answer through UART.



Serial Port Setting
Port USB Ser...(COM
Baudrate 9600
Data Bits 8
Parity None
Stop Bits 1
Flow Type None
Receive Setting
Text [✓ Hex]
[✓] Auto Feed Line
[✓] Display Send
[✓] Display Time
Send Setting
Text [✓ Hex]
Loop 1000 ms

```
21:52:56.997 0A 05
21:52:57.015 05
```

0A 05

COM3 OPENED, 9600, 8, NONE, 1, OFF   Rx: 1 Bytes   Tx: 2 Bytes

Figure 4: The CPU receives parameters 0x0A and 0x05, giving the return 0x05.

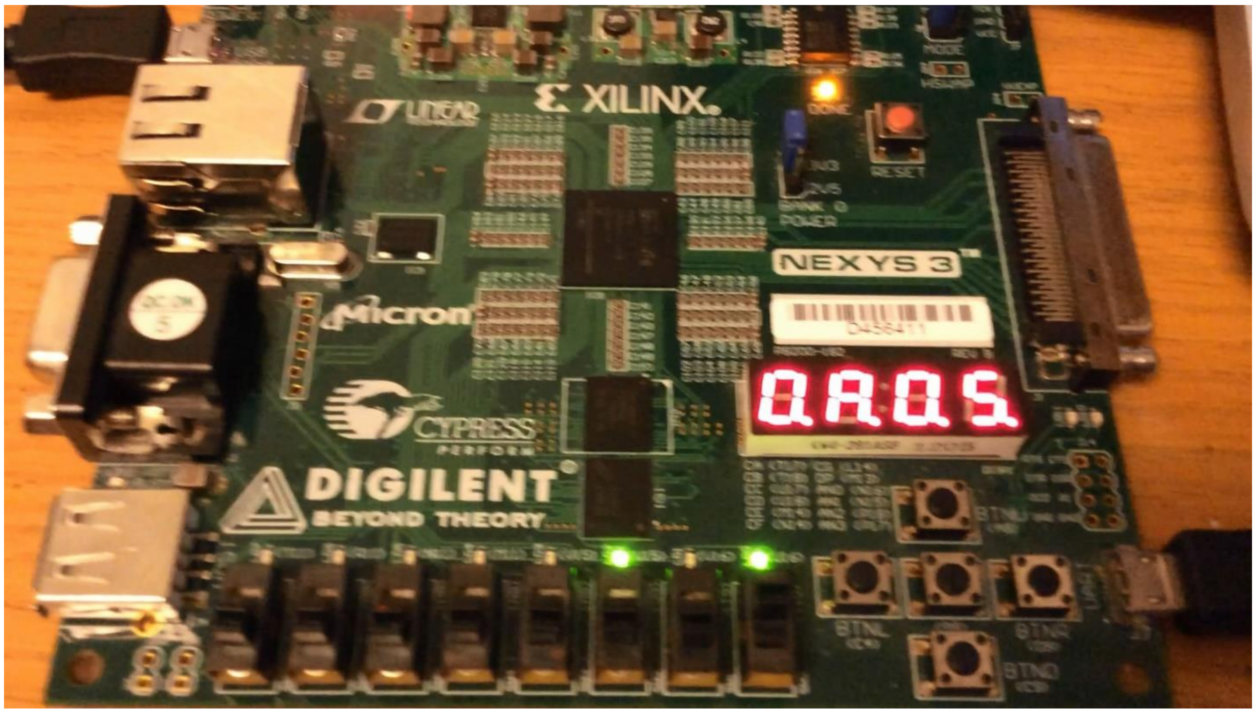


Figure 5: The LEDs are displaying the number 0x05, while the seven-segment display is showing the parameter 0x0A and 0x05.

Zhengrong Wang / 王钲荣

2012011122

EE25

Tsinghua University

Digital Logic and CPU Report

Peripherals, MIPS Code and Pipeline CPU

Peripherals

All the peripherals are mapped into an address in the memory, and the CPU visit these memory to control these peripherals. The peripherals are:

Data Memory, Timer, LEDs, Switch, Digits, UART

Data Memory:

The Data Memory serves to store the data in memory. Its address in byte is 0x00000000~0x000003FF, which is 256×32 bits. In one cycle it can read once or write once controlled by the control signal MemRead and MemWrite.

Timer:

Timer is used to generate the interrupt. During the interrupt the CPU will stop executing the user instruction and jump to some address in the instruction memory.

The timer use these following address:

0x40000000: TH

0X40000004: TL

0X40000008: TCON

When the timer is working correctly, it will increase the TL by one every cycle. And when TL reaches 0xFFFFFFF, it will load TH into TL, which means TL = TH. The tricky part is TCON as the control signal. It contains 3 bits:

0 bit: 1: Enable the timer, 0: Disable

1 bit: Control the interrupt, 1 enable and 0 disable

2 bit: Interrupt state.

And here is the Verilog code for the timer:

```
if (TCON[0]) begin
      if (TL == 32'hffffffff) begin
            TL <= TH;
            if (TCON[1]) begin
                  // give an interrupt
                  TCON[2] <= 1'b1;
            end
      end else begin
            TL <= TL + 32'b1;
      end
end
```

And if you want to use the timer, you should do this in the MIPS code:

```
addi $t1, $zero, 3
# -24($s0) is 0x40000008
sw    $t1, -24($s0)
```

During the interrupt, you should at first disable the timer, and after the interrupt is handled, enable the timer again.

```
addi $t3, $zero, 0
sw $t3, -24($s0)
```

LEDs

There are 8 LEDs on Nexys3, which are mapped into 0x4000000C, the lower 8 bits are used to represent the LEDs.

Switches

There are 8 switches on Nexys3, which are mapped into 0x40000010, the lower 8 bits are used to represent the switches.

Digits

The digits are mapped into 0x40000014, the lower 8 bits are the decoded result. And since the Nexys3 uses scan to show the digits. The 8 ~ 11 bits are the enable signals.

UART

This is definitely the most complex peripherals in the project. We use the UART designed before, which has been introduced in detail in another report. So here is the briefly introduction:

0x40000018: UART_TXD

0x4000001C: UART_RXD

0x40000020: UART_CON

The UART_TXD is the data UART is going to send to the serial, while the UART_RXD is the data UART has received from the serial. This is just based on the lab instruction file. However I have designed the UART_CON signal by myself. It contains 3 bits:

0 bit: TX_EN: an impulse will trigger the UART to send the data in UART_TXD.

1 bit: RX_EFF: 1 when the data in UART_RXD is unused.

2 bit: TX_STATUS: 1 when the UART Sender is available. 0 when it is busy.

So to read the UART, the MIPS code is like this:

```
READ_LOOP_1:
     # this loop is to read the first parameter from UART
     # 0($s0) is 0x40000020
     lw  $t0, 0($s0)
     sll $t0, $t0, 30
     srl $t0, $t0, 31      # $t0: 0 bit is RX_EFF
     bne $t0, $zero, EXIT_READ_LOOP_1
     j READ_LOOP_1

EXIT_READ_LOOP_1:
     # get the first parameter into $a0
     lw  $s1, -4($s0)
```

And in order to use UART to send some data, the MIPS code goes like this:

```
     # save the result to UART_TXD
     sw  $s3, -8($s0)
SEND_LOOP:
     # this loop is to send the result through UART
     lw  $t0, 0($s0)
     sll $t0, $t0, 29
     srl $t0, $t0, 31      # $t0: 0 bit is TX_STATUS
     bne $t0, $zero, SEND
     j SEND_LOOP
```

```
SEND:
     # now the UART_Sender is available
     # set TX_EN to send the result in UART_TXD
     addi $t0, $zero, 1
     sw   $t0, 0($s0)
     sw    $zero, 0($s0)
     j READ_LOOP_1
```

The Structure of Peripherals

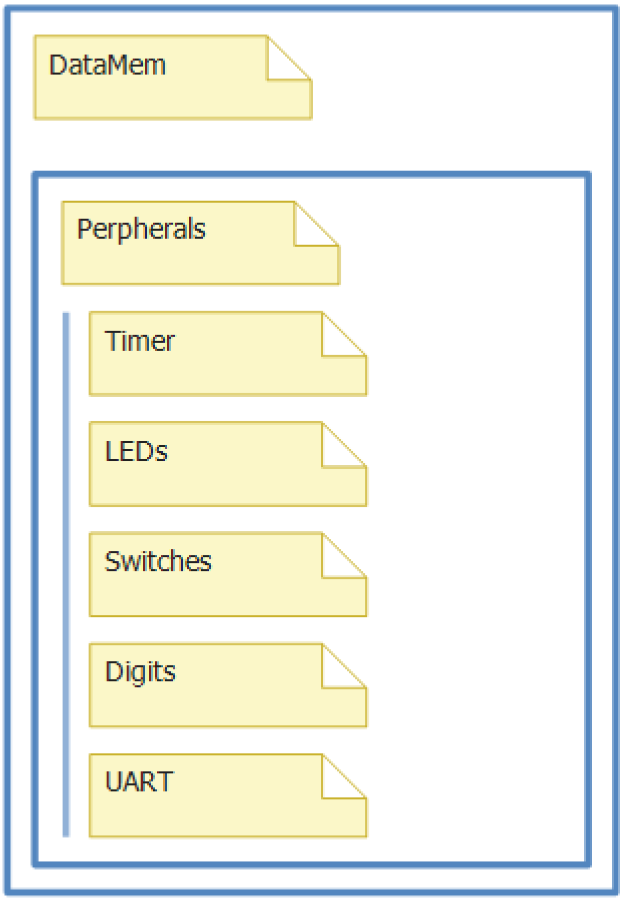
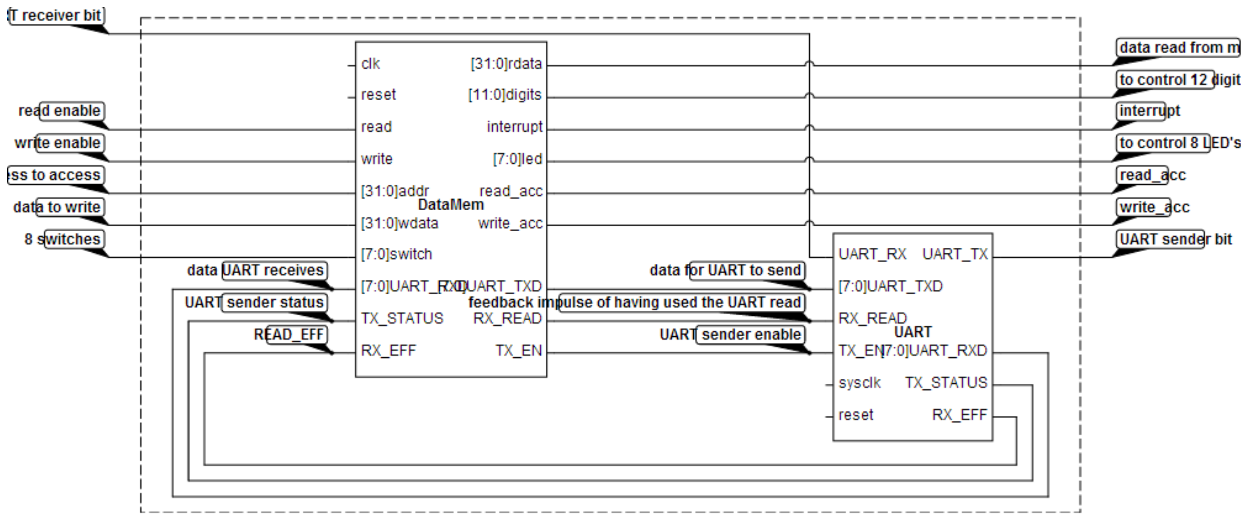


Figure1: The structure of peripherals

Figure 1 shows the structure of UART. All the peripherals are capsuled in the module Peripherals, and then within the DataMem as the whole memory.



A piece of code: the DataMem module

Here we present the actual code of the DataMem module.

```
`timescale 1ns / 1ps

// RAM Module
// Created by Zhengrong Wang
// Created 02/07/2014
// Last Modified 05/07/2014 by Zhixun Tan

// Modification note:
// 1. Modify indentation: there are only 4-spaces now
// 2. I've created a graph showing the connection between DataMem and
UART

module DataMem(
    input clk,
    input reset,
    input read,
    input write,
    input [31:0] addr,
    output reg [31:0] rdata,
    input [31:0] wdata,
    output [7:0] led,
    input [7:0] switch,
    output [11:0] digits,
    output [7:0] UART_TXD,
    input [7:0] UART_RXD,
    input TX_STATUS,
    input RX_EFF,
    output TX_EN,
    output RX_READ,
    output interrupt,
    output reg read_acc,
    output reg write_acc
);

    // the peripheral instance
    wire [31:0] peripheral_rdata;
    wire peripheral_racc;
    wire peripheral_wacc;
    Peripheral peripheral_inst(
        .reset(reset),
        .clk(clk),
        .read(read),
        .write(write),
        .addr(addr),
        .wdata(wdata),
        .rdata(peripheral_rdata),
        .led(led),
        .switch(switch),
        .digits(digits),
        .UART_TXD(UART_TXD),
        .UART_RXD(UART_RXD),
        .TX_STATUS(TX_STATUS),
        .RX_EFF(RX_EFF),
        .TX_EN(TX_EN),
        .RX_READ(RX_READ),
        .read_acc(peripheral_racc),
        .write_acc(peripheral_wacc),
        .interrupt(interrupt)
    );

    // gloabal size
    parameter RAM_SIZE = 16;
    integer i;
    reg [31:0] RAM_DATA[RAM_SIZE-1:0];

    // stack size
    parameter STACK_SIZE = 16;
    reg [31:0] STACK_DATA[10'h3ff:10'h3ff - STACK_SIZE + 1];
    wire [1:0] addr_lower;
    wire [11:2] addr_word_align; // word align addr
    wire [31:12] addr_upper;
    wire [9:0] addr_eff;
    // split the address into three parts
    // addr_lower to check if the address is word aligned, normally it
should be 2'b00
    // addr_eff is 10 bits, this is the effect word address, and it
can be extended
    // addr_upper is higher bits, used to tell whether data or
peripherals should be read

    assign addr_lower = addr[1:0];
    assign addr_eff = addr[11:2];
    assign addr_upper = addr[31:12];

    always @(*) begin
        read_acc = 1'b0;
        rdata = 32'hcccccccc;

        if (read == 1'b1) begin

            // check if the address is word aligned
            if (addr_lower == 2'b00) begin
                case(addr_upper)

                // read the data
                20'b0: begin
                    if (addr_eff < RAM_SIZE) begin
                        rdata = RAM_DATA[addr_eff];
                        read_acc = 1'b1;
                    end
                end

                // read the stack
                20'b0011_1111_1111_1111_1111: begin
                    if (addr_eff > 10'h3ff - STACK_SIZE) begin
                        rdata = STACK_DATA[addr_eff];
                        read_acc = 1'b1;
                    end
                end

                // read the peripheral
                20'b0100_0000_0000_0000_0000: begin
                    read_acc = peripheral_racc;
                    if (read_acc) begin
                        rdata = peripheral_rdata;
                    end
                end

                default: begin
                    read_acc = 1'b0;
                    rdata = 32'hcccccccc;
                end

                endcase
            end

        end

    end

    // to write the RAM
    always @(posedge clk or negedge reset) begin
        if (~reset) begin

            // initialize everything to zero

            for (i = 0; i < RAM_SIZE; i = i + 1) begin
                RAM_DATA[i] <= 32'h0;
            end

            for (i = 0; i < STACK_SIZE; i = i + 1) begin
                STACK_DATA[10'h3ff - i] <= 32'h0;
            end

            write_acc <= 1'b0;

        end else if (write == 1'b1) begin
            write_acc <= 1'b0;

            // check if the address is word aligned
            if (addr_lower == 2'b00) begin
                case (addr_upper)

                // write the data
                20'b0: begin
                    if (addr_eff < RAM_SIZE) begin
```

```
                        RAM_DATA[addr_eff] <= wdata;
                        write_acc <= 1'b1;
                    end
                end

                // write the stack
                20'b0011_1111_1111_1111_1111: begin
                    if (addr_eff > 10'h3ff - STACK_SIZE) begin
                        STACK_DATA[addr_eff] <= wdata;
                        write_acc <= 1'b1;
                    end
                end

                // write the peripherals
                20'b0100_0000_0000_0000_0000: begin
                    write_acc <= peripheral_wacc;
                end
                endcase
            end

        end
    end

endmodule
```

File List

File name	*File Description*
DataMem.v	Data Memory module
divider_16.v	Divider 16
Peripheral.v	Peripheral module
regfile.v	Provided code for regfile
UART.v	UART top module
UART_Baud_Rate_Generator.v	Generate the baud rate clock
UART_Receiver.v	UART Receiver module
UART_Sender.v	UART Sender module

The MIPS Code

The MIPS code will receive two 8 bits number from UART send their greatest common divisor through UART. BTW the program will use the digits to show these two numbers and the LEDs to show the greatest common divisor. The read and send from UART has been discussed above, hence here we only present the algorithm to find the greatest common divisor and the decoder for the digits.

Find the Greatest Common Divisor

We use the Euclidean Algorithm to find the greatest common divisor. The main part of this algorithm is a loop. The process receives two parameter in $a0 and $a1, and return the greatest common divisor in $v0.

The code is pasted here:

```
Euclidean:
      # save $s0, $s1 to the stack
      addi $sp, $sp, -8
      sw $s0, 0($sp)
      sw $s1, 4($sp)

      # load $a0, $a1 into $s0, $s1
      add $s0, $a0, $zero
      add $s1, $a1, $zero
LOOP:
      slt $t0, $s0, $s1
      beq $t0, $zero, NOT_SWITCH

      # switch $s0 and $s1 to make sure $s0 > $s1
      add $t0, $s0, $zero
      add $s0, $s1, $zero
      add $s1, $t0, $zero

NOT_SWITCH:
      sub $t0, $s0, $s1

      # if $t0 == 0, then we have found the greatest common divisor
      beq $t0, $zero, Euclidean_RETURN
      add $s0, $t0, $zero
      j LOOP

Euclidean_RETURN:
      add $v0, $s0, $zero
      lw $s1, 4($sp)
```

```
    lw $s0, 0($sp)
    addi $sp, $sp, 8
    jr $ra
```

Decoder for digits

This program is used to decode the number to show in digits. It's just a lot of if else structure. And here we present the code:

```
DECODER:
    # take in an 4 bits number in $a0
    # decode it to an 7 bits DIGITs
    # retrun it in $v0

    # 0 -> 000_0001 (1)
    bne $a0, $zero, DECODER_NOT_0
    addi $v0, $zero, 1
    j DECODER_RETURN

DECODER_NOT_0:
    addi $t0, $zero, 1
    bne $a0, $t0, DECODER_NOT_1

    # 1 -> 100_1111 (79)
    addi $v0, $zero, 79
    j DECODER_RETURN

DECODER_NOT_1:
    addi $t0, $zero, 2
    bne $a0, $t0, DECODER_NOT_2

    # 2 -> 001_0010 (18)
    addi $v0, $zero, 18
    j DECODER_RETURN

DECODER_NOT_2:
    addi $t0, $zero, 3
    bne $a0, $t0, DECODER_NOT_3

    # 3 -> 000_0110 (6)
    addi $v0, $zero, 6
    j DECODER_RETURN

DECODER_NOT_3:
    addi $t0, $zero, 4
```

```
      bne $a0, $t0, DECODER_NOT_4

      # 4 -> 100_1100 (76)
      addi $v0, $zero, 76
      j DECODER_RETURN

DECODER_NOT_4:
      addi $t0, $zero, 5
      bne $a0, $t0, DECODER_NOT_5

      # 5 -> 010_0100 (36)
      addi $v0, $zero, 36
      j DECODER_RETURN

DECODER_NOT_5:
      addi $t0, $zero, 6
      bne $a0, $t0, DECODER_NOT_6

      # 6 -> 010_0000 (32)
      addi $v0, $zero, 32
      j DECODER_RETURN

DECODER_NOT_6:
      addi $t0, $zero, 7
      bne $a0, $t0, DECODER_NOT_7

      # 7 -> 000_1111 (15)
      addi $v0, $zero, 15
      j DECODER_RETURN

DECODER_NOT_7:
      addi $t0, $zero, 8
      bne $a0, $t0, DECODER_NOT_8

      # 8 -> 000_0000 (0)
      addi $v0, $zero, 0
      j DECODER_RETURN

DECODER_NOT_8:
      addi $t0, $zero, 9
      bne $a0, $t0, DECODER_NOT_9

      # 9 -> 000_0100 (4)
      addi $v0, $zero, 4
```

```
     j DECODER_RETURN

DECODER_NOT_9:
     addi $t0, $zero, 10
     bne $a0, $t0, DECODER_NOT_A

     # A -> 000_1000 (8)
     addi $v0, $zero, 8
     j DECODER_RETURN

DECODER_NOT_A:
     addi $t0, $zero, 11
     bne $a0, $t0, DECODER_NOT_B

     # B -> 110_0000 (96)
     addi $v0, $zero, 96
     j DECODER_RETURN

DECODER_NOT_B:
     addi $t0, $zero, 12
     bne $a0, $t0, DECODER_NOT_C

     # C -> 011_0001 (49)
     addi $v0, $zero, 49
     j DECODER_RETURN

DECODER_NOT_C:
     addi $t0, $zero, 13
     bne $a0, $t0, DECODER_NOT_D

     # D -> 100_0010 (66)
     addi $v0, $zero, 66
     j DECODER_RETURN

DECODER_NOT_D:
     addi $t0, $zero, 14
     bne $a0, $t0, DECODER_NOT_E

     # E -> 011_0000 (48)
     addi $v0, $zero, 48
     j DECODER_RETURN

DECODER_NOT_E:
     # F -> 011_1000 (56)
```

```
        addi $v0, $zero, 56
        j DECODER_RETURN

DECODER_RETURN:
        jr $ra
```

The interrupt

The MIPS will use the interrupt to scan the digits, thus we need a register to record which digit is current being scanned. Here we use $t7. And the program is just use $t7 to tell which digit to scan and modify $t7 to the next digit.

The code is represented here:

```
INTERRUPT:
        # use the DIGITs to show the parameters
        bne  $t7, $zero, INTERRUPT_NOT_0
        sw    $s4, -12($s0)
        addi $t7, $zero, 1
        j KERNAL_RETURN

INTERRUPT_NOT_0:
        addi $t0, $zero, 1
        bne  $t7, $t0, INTERRUPT_NOT_1
        sw    $s5, -12($s0)
        addi $t7, $zero, 2
        j KERNAL_RETURN

INTERRUPT_NOT_1:
        addi $t0, $zero, 2
        bne  $t7, $t0, INTERRUPT_NOT_2
        sw    $s6, -12($s0)
        addi $t7, $zero, 3
        j KERNAL_RETURN

INTERRUPT_NOT_2:
        sw    $s7, -12($s0)
        addi $t7, $zero, 0
        j KERNAL_RETURN
```

After scanned the digits, the program will jump back to the address stored in $k0, and continue to execute the user instruction.

```
KERNAL_RETURN:
        jr $k0
```

File List

File name	*File Description*
main.s	The MIPS program

The Pipeline CPU

The pipeline CPU is basically just divide the single-cycle CPU into five stages and each stage will execute different instruction. It shares almost the same control unit as the single cycle CPU. However the pipeline structure will bring some hazard and need some special module to handle, which we will discuss in detail.

Regfile

The code provided for the Regfile module will write every posedge of the clock, however this design need a new forward as WB to ID stage forward, which will make the forward unit much more complex. Therefore I slightly modify the code in Regfile and make it write at every negedge of the clock.

The code is here:

```
always@(negedge reset or negedge clk) begin
      if(~reset) begin
            for(i = 1;i < 32;i = i + 1) begin
                  if (i == 29) begin
                        RF_DATA[i] <= STACK_BOTTOM;
                  end
                  else begin
                        RF_DATA[i] <= 32'b0;
                  end
            end
      end
      else begin
            if(wr && addr3) RF_DATA[addr3] <= data3;
      end
end
```

Forward

There is a Forward module to check if the data need forward. Basically it will check the register in EX and MEM stage, and then generate the control signal for two forward multiplexor to forward the data.

```
`timescale 1ns / 1ps
module Forward_Unit(
      input EX_MEM_RegWr,
      input [4:0] EX_MEM_RegDst,
      input [4:0] ID_EX_Rt,
      input [4:0] ID_EX_Rs,
      input [2:0] ID_PCSrc,
      input [4:0] IF_ID_Rd,
      input [4:0] ID_EX_Rd,
      input ID_EX_RegWr,
      input MEM_WB_RegWr,
      input [4:0] MEM_WB_RegDst,
      output reg [1:0] ForwardA,
      output reg [1:0] ForwardB,
      output reg [1:0] ForwardJr
     );

always @(*) begin
      if (EX_MEM_RegWr == 1'b1 &&
            EX_MEM_RegDst != 5'h00 &&
            EX_MEM_RegDst == ID_EX_Rs) begin
            ForwardA = 2'b10;
      end
      else if (MEM_WB_RegWr == 1'b1 &&
                  MEM_WB_RegDst != 5'h00 &&
                  MEM_WB_RegDst == ID_EX_Rs) begin
            ForwardA = 2'b01;
      end
      else begin
            ForwardA = 2'b00;
      end
      if (EX_MEM_RegWr == 1'b1 &&
            EX_MEM_RegDst != 5'h00 &&
            EX_MEM_RegDst == ID_EX_Rt) begin
            ForwardB = 2'b10;
      end
      else if (MEM_WB_RegWr == 1'b1 &&
                  MEM_WB_RegDst != 5'h00 &&
                  MEM_WB_RegDst == ID_EX_Rt) begin
            ForwardB = 2'b01;
      end
      else begin
```

```
            ForwardB = 2'b00;
        end
        if (ID_PCSrc == 3'b011 &&
            IF_ID_Rd == ID_EX_Rd &&
            ID_EX_Rd != 0 && ID_EX_RegWr) begin
            ForwardJr = 2'b01;
        end
        else if (ID_PCSrc == 3'b011 &&
                IF_ID_Rd != ID_EX_Rd &&
                IF_ID_Rd == EX_MEM_RegDst &&
                EX_MEM_RegWr &&
                EX_MEM_RegDst != 0) begin
            ForwardJr = 2'b10;
        end
        else if (ID_PCSrc == 3'b011 &&
                IF_ID_Rd != ID_EX_Rd &&
                IF_ID_Rd != EX_MEM_RegDst &&
                IF_ID_Rd == MEM_WB_RegDst &&
                MEM_WB_RegDst != 0 &&
                MEM_WB_RegWr) begin
            ForwardJr = 2'b11;
        end
        else begin
            ForwardJr = 2'b00;
        end
end
endmodule
```

Hazard

There is still some hazard cannot be solved by forward, for example the load-use hazard. And here is the hazard detection unit. It will detect the hazard and generate two control signals: Write and Flush. Write will control the stage register either to write new data in at the posedge of clock or not, which means it can maintain the instruction not to flow in the pipeline. This is basically used in load—use hazard to generate the stalk. Flush signal, when it's enabled, will reset the corresponding stage register, which means canceling the instruction in that stage. This is used in the branch and jump hazard.

The code is here:

```
`timescale 1ns / 1ps

// ZHENGRONG WANG
// Created 11/07/2014
// Last Modified 14/07/2014

module Hazard_Detection_Unit(
     input ID_EX_MemRd,
     input [4:0] ID_EX_Rt,
     input [4:0] IF_ID_Rs,
     input [4:0] IF_ID_Rt,
     input [2:0] ID_PCSrc,
     input [2:0] ID_EX_PCSrc,
     input EX_ALUResult0,
     output reg [2:0] PCWrite,
     output reg [2:0] IF_ID_WRITE,
     output reg [2:0] IF_ID_Flush,
     output reg [2:0] ID_EX_Flush
    );

always @(*) begin
     if (ID_EX_MemRd == 1'b1 &&
          (ID_EX_Rt == IF_ID_Rs || ID_EX_Rt == IF_ID_Rt)) begin
          PCWrite[0] = 1'b0;
          IF_ID_WRITE[0] = 1'b0;
          ID_EX_Flush[0] = 1'b0;
          IF_ID_Flush[0] = 1'b1;
     end
     else begin
          PCWrite[0] = 1'b1;
          IF_ID_WRITE[0] = 1'b1;
          ID_EX_Flush[0] = 1'b1;
          IF_ID_Flush[0] = 1'b1;
     end
end

always @(*) begin
     if (ID_PCSrc == 3'b010 ||
          ID_PCSrc == 3'b011 ||
          ID_PCSrc == 3'b100 ||
          ID_PCSrc == 3'b101) begin
          PCWrite[1] = 1'b1;
          IF_ID_WRITE[1] = 1'b1;
          ID_EX_Flush[1] = 1'b1;
          IF_ID_Flush[1] = 1'b0;
     end
     else begin
          PCWrite[1] = 1'b1;
```

```
            IF_ID_WRITE[1] = 1'b1;
            ID_EX_Flush[1] = 1'b1;
            IF_ID_Flush[1] = 1'b1;
      end
end

always @(*) begin
      if (ID_EX_PCSrc == 3'b001 &&
            EX_ALUResult0 == 1'b1) begin
            PCWrite[2] = 1'b1;
            IF_ID_WRITE[2] = 1'b1;
            ID_EX_Flush[2] = 1'b0;
            IF_ID_Flush[2] = 1'b0;
      end
      else begin
            PCWrite[2] = 1'b1;
            IF_ID_WRITE[2] = 1'b1;
            ID_EX_Flush[2] = 1'b1;
            IF_ID_Flush[2] = 1'b1;
      end
end

endmodule
```

File List

File name	*File Description*
Control_Unit.v	Control module
EX_MEM_REG.v	EX/MEM Register
Extend.v	Extend module
Forward_Unit.v	Forward Module
Hazard_Detection_Unit.v	Hazard Detection Module
ID_EX_REG.v	ID/EX Register
IF_ID_REG.v	IF/ID Register
MEM_WB_REG.v	MEM/WB Register
MUX.v	Multiplexor(2 bits)
MUX2.v	Multiplexor(1 bits)
MUX8.v	Multiplexor(3 bits)
PC_REG.v	PC Register

Pipeline_Core.v	Pipeline Core module
Pipeline_fpga.bit	Bit file generated
Pipeline_fpga.v	Pipeline fpga module
Pipeline_Regfile.v	Pipeline Regfile module
rom.v	Rom module contains the instruction

XianYu Meng / 孟宪妤

2012011116

EE25

Tsinghua University

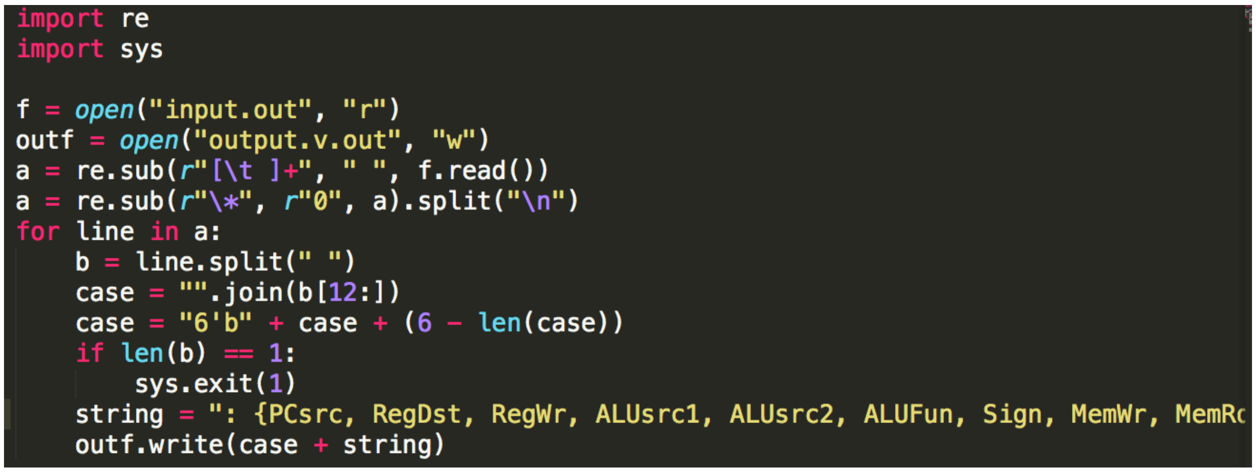
Digital Logic and CPU Report

Main control unit ,Ext32 and PC unit

Control unit

To start this process, we need to identify the fields of an instruction and the control lines that are needed for the datapath.

To produce the control signal, I've wrote a piece of python code which is shown below; utilize the excel (be outputted as a txt) to generate all the signals.



```
import re
import sys

f = open("input.out", "r")
outf = open("output.v.out", "w")
a = re.sub(r"[\t ]+", " ", f.read())
a = re.sub(r"\*", r"0", a).split("\n")
for line in a:
    b = line.split(" ")
    case = "".join(b[12:])
    case = "6'b" + case + (6 - len(case))
    if len(b) == 1:
        sys.exit(1)
    string = ": {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2, ALUFun, Sign, MemWr, MemRc
    outf.write(case + string)
```

import re

import sys

f = open("input.out", "r")

outf = open("output.v.out", "w")

a = re.sub(r"[\t]+", " ", f.read())

a = re.sub(r"*", r"0", a).split("\n")

for line in a:

```
        b = line.split(" ")
        case = "".join(b[12:])
        case = "6'b" + case + (6 - len(case))
        if len(b) == 1:
                sys.exit(1)
        string = ": {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2, ALUFun, Sign, MemWr, MemRd,
MemtoReg, EXTOp, LUOp} <= {3\'h%s, 2\'h%s, 1\'h%s, 1\'h%s, 1\'h%s, 6\'%s, 1\'h%s, 1\'h%s,
1\'h%s, 2\'h%s, 1\'h%s, 1\'h%s};\n"%tuple(b[:12])
        outf.write(case + string)



module control(
        instruct,
        supervisor, //if(supervisor) exceptions=0 else exceptions=1
        PCsrc,
        RegDst,
        RegWr,
        ALUsrc1,
        ALUsrc2,
        ALUFun,
        Sign,
        MemWr,
        MemRd,
        MemtoReg,
        EXTOp,
        LUOp);
        input reg [31:0] instruct;
```

```
	input reg supervisor;
	output reg [2:0] PCsrc;
	output reg [1:0] RegDst;
	output reg RegWr;
	output reg ALUsrc1;
	output reg ALUsrc2;
	output reg [5:0] ALUFun;
	output reg Sign;
	output reg MemWr;
	output reg MemRd;
	output reg [1:0] MemtoReg;
	output reg EXTOp;
	output reg LUOp;
	parameter ALU_ARITH = 2'b00;
parameter ALUFUNC_ADD = 6'b000000;
parameter ALUFUNC_SUB = 6'b000001;

parameter ALU_LOGIC = 2'b01;
parameter ALUFUNC_AND = 6'b011000;
parameter ALUFUNC_OR  = 6'b011110;
parameter ALUFUNC_XOR = 6'b010110;
parameter ALUFUNC_NOR = 6'b010001;
parameter ALUFUNC_A   = 6'b011010;

parameter ALU_SHIFT = 2'b10;
parameter ALUFUNC_SLL = 6'b100000;
parameter ALUFUNC_SRL = 6'b100001;
parameter ALUFUNC_SRA = 6'b100011;


parameter ALU_CMP = 2'b11;
parameter ALUFUNC_EQ  = 6'b110011;
parameter ALUFUNC_NEQ = 6'b110001;
parameter ALUFUNC_LT  = 6'b110101;
parameter ALUFUNC_LEZ = 6'b111101;
parameter ALUFUNC_GEZ = 6'b111001;
parameter ALUFUNC_GTZ = 6'b111111;


    always @(*)
        begin
        if (IRQsig && !supervisor)
            {PCsrc, RegDst, RegWr, MemtoReg} <= {3'h4, 2'h3, 1'h1, 2'h2};
        end


        else begin
            case (instruct[31:26]) //opcode
                6'b000000:begin
                    //R-format instructions, which all have an opcode of 0.
        //three register operands:rs,rt,rd.
        //rs,rt are sources;rd is the destinations.
        //The ALU functions is in the funct field and is decoded by the ALU control.
                    case (instruct[5:0]) //Funct
                        6'b100000:begin //add
```

```
                                        {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2,
ALUFun, Sign, MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h0, 2'h0, 1'h1, 1'h0, 1'h0,
ALUFUNC_ADD, 1'h1, 1'h0, 1'h0, 2'h0, 1'h0, 1'h0};

                                end

                                6'b100001:begin //addu

                                        {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2,
ALUFun, Sign, MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h0, 2'h0, 1'h1, 1'h0, 1'h0,
ALUFUNC_ADD, 1'h0, 1'h0, 1'h0, 2'h0, 1'h0, 1'h0};

                                end

                                6'b100010:begin //sub

                                        {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2,
ALUFun, Sign, MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h0, 2'h0, 1'h1, 1'h0, 1'h0,
ALUFUNC_SUB, 1'h1, 1'h0, 1'h0, 2'h0, 1'h0, 1'h0};

                                end

                                6'b100011:begin //subu

                                        {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2,
ALUFun, Sign, MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h0, 2'h0, 1'h1, 1'h0, 1'h0,
ALUFUNC_SUB, 1'h0, 1'h0, 1'h0, 2'h0, 1'h0, 1'h0};

                                end

                                6'b100100:begin //and

                                        {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2,
ALUFun, Sign, MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h0, 2'h0, 1'h1, 1'h0, 1'h0,
ALUFUNC_AND, 1'h0, 1'h0, 1'h0, 2'h0, 1'h0, 1'h0};

                                end

                                6'b100101:begin //or

                                        {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2,
ALUFun, Sign, MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h0, 2'h0, 1'h1, 1'h0, 1'h0,
ALUFUNC_OR, 1'h0, 1'h0, 1'h0, 2'h0, 1'h0, 1'h0};

                                end

                                6'b100110:begin //xor
```

```
                                        {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2,
ALUFun, Sign, MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h0, 2'h0, 1'h1, 1'h0, 1'h0,
ALUFUNC_XOR, 1'h0, 1'h0, 1'h0, 2'h0, 1'h0, 1'h0};
                                end
                                6'b100111:begin //nor
                                        {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2,
ALUFun, Sign, MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h0, 2'h0, 1'h1, 1'h0, 1'h0,
ALUFUNC_NOR, 1'h0, 1'h0, 1'h0, 2'h0, 1'h0, 1'h0};
                                end
                                6'b101010:begin //slt
                                        {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2,
ALUFun, Sign, MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h0, 2'h0, 1'h1, 1'h0, 1'h0,
ALUFUNC_LT, 1'h1, 1'h0, 1'h0, 2'h0, 1'h0, 1'h0};
                                end
                                6'b101011:begin //sltu
                                        {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2,
ALUFun, Sign, MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h0, 2'h0, 1'h1, 1'h0, 1'h0,
ALUFUNC_LT, 1'h0, 1'h0, 1'h0, 2'h0, 1'h0, 1'h0};
                                end
                                6'b000000:begin //sll
                                        {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2,
ALUFun, Sign, MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h0, 2'h0, 1'h1, 1'h1, 1'h0,
ALUFUNC_SLL, 1'h0, 1'h0, 1'h0, 2'h0, 1'h0, 1'h0};
                                end
                                6'b000010:begin //srl
                                        {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2,
ALUFun, Sign, MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h0, 2'h0, 1'h1, 1'h1, 1'h0,
ALUFUNC_SRL, 1'h0, 1'h0, 1'h0, 2'h0, 1'h0, 1'h0};
                                end
                                6'b000011:begin //sra
```

```
                                                {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2,
ALUFun, Sign, MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h0, 2'h0, 1'h1, 1'h1, 1'h0,
ALUFUNC_SRA, 1'h0, 1'h0, 1'h0, 2'h0, 1'h0, 1'h0};
                                        end
                                        6'b001000:begin //jr
                                                {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2,
ALUFun, Sign, MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h3, 2'h0, 1'h0, 1'h0, 1'h0,
6'bxxxxxx, 1'h0, 1'h0, 1'h0, 2'h0, 1'h0, 1'h0};
                                        end
                                        6'b001001:begin //jalr
                                                {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2,
ALUFun, Sign, MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h3, 2'h2, 1'h1, 1'h0, 1'h0,
6'bxxxxxx, 1'h0, 1'h0, 1'h0, 2'h2, 1'h0, 1'h0};
                                        end
                                                default : /* default */;
                                endcase
                        6'b001000:begin //addi
                                {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2, ALUFun, Sign,
MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h0, 2'h1, 1'h1, 1'h0, 1'h1, ALUFUNC_ADD,
1'h1, 1'h0, 1'h0, 2'h0, 1'h1, 1'h0};
                        end
                        6'b001001:begin //addiu
                                {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2, ALUFun, Sign,
MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h0, 2'h1, 1'h1, 1'h0, 1'h1, ALUFUNC_ADD,
1'h0, 1'h0, 1'h0, 2'h0, 1'h0, 1'h0};
                        end
                        6'b001010:begin //slti
                                {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2, ALUFun, Sign,
MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h0, 2'h1, 1'h1, 1'h0, 1'h1, ALUFUNC_LT,
1'h1, 1'h0, 1'h0, 2'h0, 1'h1, 1'h0};
                        end
```

```
                    6'b001011:begin //sltiu
                        {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2, ALUFun, Sign,
MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h0, 2'h1, 1'h1, 1'h0, 1'h1, ALUFUNC_LT,
1'h0, 1'h0, 1'h0, 2'h0, 1'h0, 1'h0};
                    end
                    6'b001100:begin //andi
                        {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2, ALUFun,
Sign, MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h0, 2'h1, 1'h1, 1'h0, 1'h1,
ALUFUNC_AND, 1'h0, 1'h0, 1'h0, 2'h0, 1'h1, 1'h0};
                    end
                    6'b000100:begin //beq
                        {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2, ALUFun, Sign,
MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h1, 2'h1, 1'h0, 1'h0, 1'h0, ALUFUNC_EQ,
1'h1, 1'h0, 1'h0, 2'h0, 1'h1, 1'h0};
                    end
                    6'b000101:begin //bne
                        {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2, ALUFun, Sign,
MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h1, 2'h1, 1'h0, 1'h0, 1'h0, ALUFUNC_NEQ,
1'h1, 1'h0, 1'h0, 2'h0, 1'h1, 1'h0};
                    end
                    6'b000110:begin //blez
                        {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2, ALUFun, Sign,
MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h1, 2'h1, 1'h0, 1'h0, 1'h0, ALUFUNC_LEZ,
1'h1, 1'h0, 1'h0, 2'h0, 1'h1, 1'h0};
                    end
                    6'b000111:begin //bgtz
                        {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2, ALUFun, Sign,
MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h1, 2'h1, 1'h0, 1'h0, 1'h0, ALUFUNC_GTZ,
1'h1, 1'h0, 1'h0, 2'h0, 1'h1, 1'h0};
                    end
                    6'b000001:begin //bgez
```

```
                                   {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2, ALUFun,
Sign, MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h1, 2'h1, 1'h0, 1'h0, 1'h0,
ALUFUNC_GEZ, 1'h1, 1'h0, 1'h0, 2'h0, 1'h1, 1'h0};
                            end
                            //Load or store instructions.
       //The register rs is in the base register that is added to the
       //16-bit address field to form the memory address.
                            6'b101011:begin //sw
                                   //for lw,rt is the destination register for the loaded
value.
                                  {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2, ALUFun, Sign,
MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h0, 2'h1, 1'h0, 1'h0, 1'h1, ALUFUNC_ADD,
1'h1, 1'h1, 1'h0, 2'h0, 1'h1, 1'h0};
                            end
                            6'b100011:begin //lw
                                   //for sw,rt is the source register whose value should be
stored into memory.
                                  {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2, ALUFun, Sign,
MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h0, 2'h1, 1'h1, 1'h0, 1'h1, ALUFUNC_ADD,
1'h1, 1'h0, 1'h1, 2'h1, 1'h1, 1'h0};
                            end
                            6'b001111:begin //lui
                                  {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2, ALUFun, Sign,
MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h0, 2'h1, 1'h1, 1'h0, 1'h1, ALUFUNC_A,
1'h0, 1'h0, 1'h0, 2'h0, 1'h0, 1'h1};
                            end
                            6'b000010:begin //j
                                  {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2, ALUFun, Sign,
MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h2, 2'h0, 1'h0, 1'h0, 1'h0, 6'b000000,
1'h0, 1'h0, 1'h0, 2'h0, 1'h0, 1'h0};
                            end
```

```
                                        6'b000011:begin //jal
                                                {PCsrc, RegDst, RegWr, ALUsrc1, ALUsrc2, ALUFun, Sign,
MemWr, MemRd, MemtoReg, EXTOp, LUOp} <= {3'h2, 2'h2, 1'h1, 1'h0, 1'h0, 6'b000000,
1'h0, 1'h0, 1'h0, 2'h2, 1'h0, 1'h0};
                                        end
                                        default : /* default */;
                                endcase


                        end
```

EXT32

To increase the size of a data item by replicating the high order sign bit of the original data item in the high order bits of the larger, destination data item. It’s the unit to signed-extend 16bits offset field in the instruction to a 32 bit signed value.

```
module Ext32(input [15:0] Imm16,  // assign Imm16 = instruction[15:0];
                        input EXTOp,
                        output [31:0] ExtendedImm);
        parameter EXTOP_UNSIGNED = 1'b0;
  parameter EXTOP_SIGNED   = 1'b1;
  always @(*) begin
    case (EXTOp)
    EXTOP_UNSIGNED: Imm32 = { 16'b0, Imm16 };
    EXTOP_SIGNED:   Imm32 = { {16{Imm16[15]}}, Imm16 };
    endcase
    assign ConBA = { Imm32[29:0], 2'b00 } + NewPC;
  end
endmodule
```

Program counter(PC unit)

Hold the address of the current instruction. Lastly we will need an adder to increment the PC to the address of the next instruction. To execute any instruction, we must start by fetching the instruction from the memory. To prepare for executing the next instruction, we must also increment the program counter so that it points at the next instruction,4 bytes later.

```
module PCUnit(
                input reset
                input clk,
                input [2:0] PCsrc,
                input ALUOut0,
                input [31:0] ConBA,
                input [25:0] JTaddress, //其值来 自于 J 型指令中的 26 位地址
                input [31:0] DatabusA, //寄存器跳转时,Databus A,其值取自$Ra
                output [31:0] PCplus4,
                output reg [31:0] PC,
                output supervisor// PC[31]
                );
     assign supervisor = PC[31];
     assign PCplus4 = PC+4;

     always @(posedge clk or posedge reset )
          begin
          if (reset)
                begin
                PC <= 32'h80000000;  //处理器复位后,PC 中的值应该为 0x80000000;
                end
          else
                begin
                case (PCsrc)
                        0: PC <= PCplus4;
                        1: PC <= ALUOut0?ConBA:PCplus4;
                        //条件分支时,ConBA 或 者 PC+4,根据条件分支中的条件判断结果
而定(ALUOut[0])
                        2: PC <= {PC[31:28],JTaddress,2'b00};
                        //直接跳转时,JT,其值来 自于 J 型指令中的 26 位地址
                        3: PC <= DatabusA;
                        //寄存器跳转时,Databus A,其值取自$Ra
                        4: PC <= 32'h80000004; //中断时,PC 中的值应该为 0x80000004
                        5: PC <= 32'h80000008; //异常时,PC 中的值应该为 0x80000008
                endcase
                end
        end
endmodule
```