A Single-Cycle CPU Documentation for Verilog Implementations

P4-afterclass **| 2023/11/3**

1. Supported Instructions

Instruction	`add`	`sub`	`ori`	`lw`	`sw`	`beq`	`lui`	`jal`	`jr`
Op	`000000`	`000000`	`001101`	`100011`	`101011`	`000100`	`001111`	`000011`	`000000`
Func	`100000`	`100010`							`001000`

Synchronized Reset, Overflow Not Considered

Notice:

- `nop` instruction must set control signals.
- Bit-width must be declared when declaring a variable of type `wire`.
- Remember to save the `wave` file when using ISim.

2. Modules definition

(1) Program Counter

- **Ports definition**

Port name	Direction	Width	Description
clk	input		
reset	input		
next_PC	input	[31:0]	`PC + 4/Beq/Jal/Jr ->`
PC	output	[31:0]	`-> Adder / GRF`
InstrAddr	output	[11:0]	`-> Im`

- ## Behavioral Description

```
if beq && Zero then
    PC <= PC + 4 + Sign_ext(imm16)
else if jal then
    PC <= {PC[31:28], addr26, 00}
else if jr then
    PC <= GPR[31] # $ra
else
    PC <= PC + 4
```

(2) Instruction Memory

- ## Ports Definition

Port name	Direction	Type	Description
InstrAddr	input	[11:0]	`PC ->`
Instr	output	[31:0]	`-> Splitter`

- ## Behavioral Description

```
# 4096 * 32bit, [31:0] Reg [0:4095]
Instr = IM[InstrAddr]
```

(3) Splitter

▪ Ports Definition

Port name	Direction	Type	Description
Instr	input	[31:0]	`PC ->`
Addr26	output	[25:0]	`Instr[25:0]`
Imm16	output	[15:0]	`Instr[15:0]`
func	output	[5:0]	`Instr[5:0]`
Rd	output	[4:0]	`Instr[15:11]`
Rt	output	[4:0]	`Instr[20:16]`
Rs	output	[4:0]	`Instr[25:21]`
Op	output	[5:0]	`Instr[31:26]`

▪ Behavioral Description

None.

(4) GRF

▪ Ports Definition

Port name	Direction	Type	Description
clk	input		
reset	input		
RegWrite	input		*`Signal`*
pC	input	[31:0]	`PC ->`
WD	input	[31:0]	`Ext / ALU / PC / DM ->`
A1	input	[4:0]	`Rs ->`
A2	input	[4:0]	`Rt ->`
WA	input	[4:0]	`Rt / Rd / 31 ->`
RD1	output	[31:0]	`-> ALU / PC`
RD2	output	[31:0]	`-> ALU / DM`

- **Behavioral Description**

```
RD1 <= GRF[A1]
RD2 <= GRF[A2]

if RegWrite == 1 then
    GRF[WA] <= WD
```

(5) ALU

- **Ports Definition**

Port name	Direction	Type	Description
srcA	input	[31:0]	`GRF ->`
srcB	input	[31:0]	`Ext / GRF ->`
ALUop	input	[1:0]	*`signal`*
Zero	output		`-> MUX(beq)`
Result	output	[31:0]	`-> GRF / DM`

- ### Behavioral Description

```
if ALUop == 2'b01
    Result <= A + B
else if ALUop == 2'b01
    Result <= A - B
else if ALUop == 2'b10
    Result <= A & B
else if ALUop == 2'b11
    Result <= A | B

if A - B == 0 then
    Zero <= 1'b1
else
    Zero <= 1'b0
```

(6) Data Memory

- ### Ports Definition

Port name	Direction	Type	Description
clk	input		
reset	input		
WriteData	input		`signal`
ReadData	input		`signal`
Addr	input	[31:0]	`ALU ->`
WD	input	[31:0]	`GRF ->`
pC	input	[31:0]	`PC ->`
RD	output	[31:0]	`-> GRF`

- ### Behavioral Description

```
if WriteData == 1'b1
    RD <= DM[Addr[13:2]]
else if ReadData == 1'b1
    DM[Addr[13:2]] <= WD
```

(7) CtrlUnit

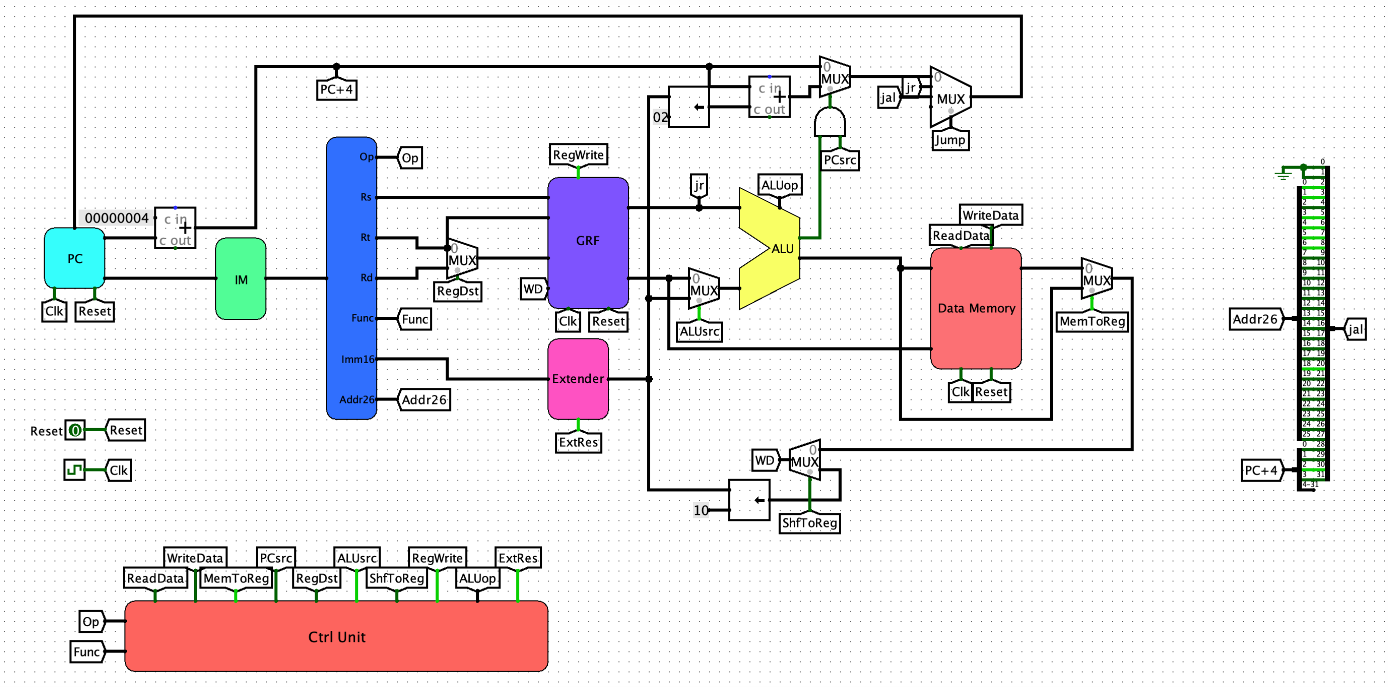
▪ Ports Definition

Port name	Direction	Type	Description
op	input	[5:0]	
func	input	[5:0]	
ReadData	output		`DM`
WriteData	output		`DM`
MemToReg	output		`GRF`
PCsrc	output		`PC`
RegDst	output		`GRF`
ALUsrc	output		`ALU`
ShfToReg	output		`GRF`
RegWrite	output		`GRF`
ALUop	output	[1:0]	`ALU`
ExtRes	output		`Extender`
Jump	output	[1:0]	`PC / GRF`

▪ Corresponding Intsructions

	RegDst	ALUsrc	ALUop[1:0]	PCsrc	ReadData	WriteData	MemToReg	ShfToReg	RegWrite	ExtRes	Jump[1:0]
`ADD`	1	0	00(Add)	0	x	x	1	0	1	x	00
`SUB`	1	0	01(Sub)	0	x	x	1	0	1	x	00
`ORI`	0	1	11(Or)	x	x	x	1	0	1	1	00
`LW`	0	1	00(Add)	0	1	x	0	0	1	0	00
`SW`	x	1	00(Add)	0	x	1	x	0	0	0	00
`BEQ`	x	0	01(Sub)	1	x	x	x	x	0	0	00
`LUI`	0	x	xx	0	x	x	x	1	1	x	00
`JAR`	0	0	xx	0	x	x	0	0	1	x	10
`JR`	x	x	xx	x	x	x	x	x	0	x	01

3. Circuit Diagram



4. Questions

（1） DM按字寻址，而不是按字节寻址。因此数据合理的情况下，保证**Addr**的后两位始终为0，即**Addr**始终是4的倍数。该信号来自ALU。

（2）控制信号每种取值所对应的指令：

```
wire ALUsrc;
assign ALUsrc = (instr == `ORI) || (instr == `LW) || (instr == `SW);

wire MemToReg;
assign MemToReg = (instr == `ADD) || (instr == `SUB) || (instr == `ORI);
```

优点：新增指令时较为方便；

缺点：不好全面掌握一种指令对应的信号，比较分散。

指令对应控制信号：

```
if (op == 6'b000000) begin                    // R
        if (func == 6'b100000) begin          // add
            RegDst = 1'b1;
            ALUsrc = 1'b0;
            ALUop = 2'b00;
            PCsrc = 1'b0;
            ReadData = 1'b0; // x
            WriteData = 1'b0; // x
```

```
                MemToReg = 1'b1;
                ShfToReg = 1'b0;
                RegWrite = 1'b1;
                ExtRes = 1'b0; // x
                Jump = 2'b00;
            end
            else if (func == 6'b100010) begin   // sub
                RegDst = 1'b1;
                ALUsrc = 1'b0;
                ALUop = 2'b01;
                PCsrc = 1'b0;
                ReadData = 1'b0; // x
                WriteData = 1'b0; // x
                MemToReg = 1'b1;
                ShfToReg = 1'b0;
                RegWrite = 1'b1;
                ExtRes = 1'b0; // x
                Jump = 2'b0;
            end
```

优点：每种指令对应的控制信号比较清晰，方便调试。

缺点：代码冗杂，新增控制信号或者新增指令时比较繁琐，可扩展性稍差。

（3）同步复位时：

```
always @(posedge clk) begin
  if (reset == 1'b1) begin
    // do something
  end
  else begin
    // do something
  end
end
```

异步复位时：（假定复位信号高电平有效）

```
always @(posedge clk or posedge reset) begin
  if (reset == 1'b1) begin
    // do something
  end
  else begin
    // do something
  end
end
```

敏感信号列表不同，在同步复位中复位信号优先级小于时钟信号，而在异步复位中两种信号优先级一致。

（4）在忽略了溢出的情况下，**add**指令不会检查运算结果是否溢出，那么指令等效于**addu**。同样的，如果忽略溢出，那么**addi**指令也不会检查结果是否溢出，这就与**addiu**本质上是相同的指令。

Appendice

Test_1

```
.text
    ori $t0, $zero, 0   # ori test
    ori $t0, $zero, 0xabcd
    ori $t1, $t0, 0x1234
    ori $t1, $zero, 12
    ori $t0, $zero, 0xffff

    lui $t0, 0xffff   # lui test
    lui $t1, 17
    lui $zero, 0xa

    lui $t0, 0
    ori $t0, $zero, 0x1234
    add $t0, $zero, $t0   # add test
    add $t0, $t0, $t0
    add $zero, $t0, $t0

    sub $t0, $t0, $zero   # sub test
    sub $t1, $t0, $t1
    sub $t1, $t1, $t1

    lui $t0, 0
    lui $t1, 0
    ori $t0, 12
    ori $t1, 0x1234
    sw $t1, 0($t0)    # sw test
    sw $t1, -4($t0)
    sw $t1, 4($t0)

    lui $t1, 0
    lw $t1, 0($t0)    # lw test
    lui $t1, 0
    lw $t1, 4($t0)
    lui $t1, 0
    lw $t1, -4($t0)

    previous:
    lui $t0, 0
    lui $t1, 0
    ori $t0, 1
    beq $t0, $t1, jump
    beq $t1, $t1, jump

    jump:
    ori $t0, $zero, 0xabcd
```

```
    beq $t0, $zero, previous
    beq $zero, $zero, previous
```

Test_2

```
.text
    jal func2

    func1:
    jr $ra

    func2:
    jal func        # jump to func and save position to $ra
    jal func1
    jr  $ra         # jump to $ra

    func:
    jr  $ra         # jump to $ra
```

Test_3

```
.text
    ori $t0, $zero, 0xabcd
    ori $t1, $zero, 0x1234

    beq $t0, $t1, func
    jal func1
    beq $zero, $zero, func
    func:
    jal func1
    func1:
    jr $ra
```

Test_4

```
.text
    ori $t0, $zero, 1
    ori $t1, $zero, 2

    add $t0, $t0, $t0
    add $t1, $t0, $t1

    sub $t1, $t1, $t0
    sub $t0, $t0, $zero

    jal func
```

```
beq $zero, $zero, branch

func:
lui $t0, 0
lui $t1, 0
ori $t1, 0x1234
sw $t1, 4($t0)
lw $t0, 4($t0)
jr $ra

branch:
ori $t0, $zero, $zero
```