H1 单周期cpu设计

PC	IM	Adder	Nadder	RF	ALU	Zeroext	Signext	DM	Sll	Comparator									
A	B	A	B	RA1	RA2	WA	WD	Op	A	B	DA	DD	Num	A					
add	Adder	PC	PC	4	\	[6:10]	[11:15]	[16:20]	ALU	+	RD1	RD2	\	\	\	\	\		
sub	Adder	PC	PC	4	\	[6:10]	[11:15]	[16:20]	ALU	-	RD1	RD2	\	\	\	\	\		
ori	Adder	PC	PC	4	\	[6:10]	\	[11:15]	ALU	|	RD1	Zeronext	[16:31]	\	\	\	\		
lw	Adder	PC	PC	4	\	[6:10]	\	[11:15]	DM	+	RD1	Signext	\	[16:31]	ALU	\	\	\	
sw	Adder	PC	PC	4	\	[6:10]	[11:15]	\	\	+	RD1	Signext	\	[16:31]	ALU	RD2	\	\	
beq	Adder|Nadder	PC	PC	4	Adder	Signext	[6:10]	[11:15]	\	\	-	RD1	RD2	\	Sll	\	2	[16:31]	ALU
lui	Adder	PC	PC	4	\	\	\	[11:15]	Sll	\	[16:31]	\	\	16	Zeroext	\			
nop	Adder	PC	PC	4	\	\	\	\	\	\	\	\							

H2 def.v

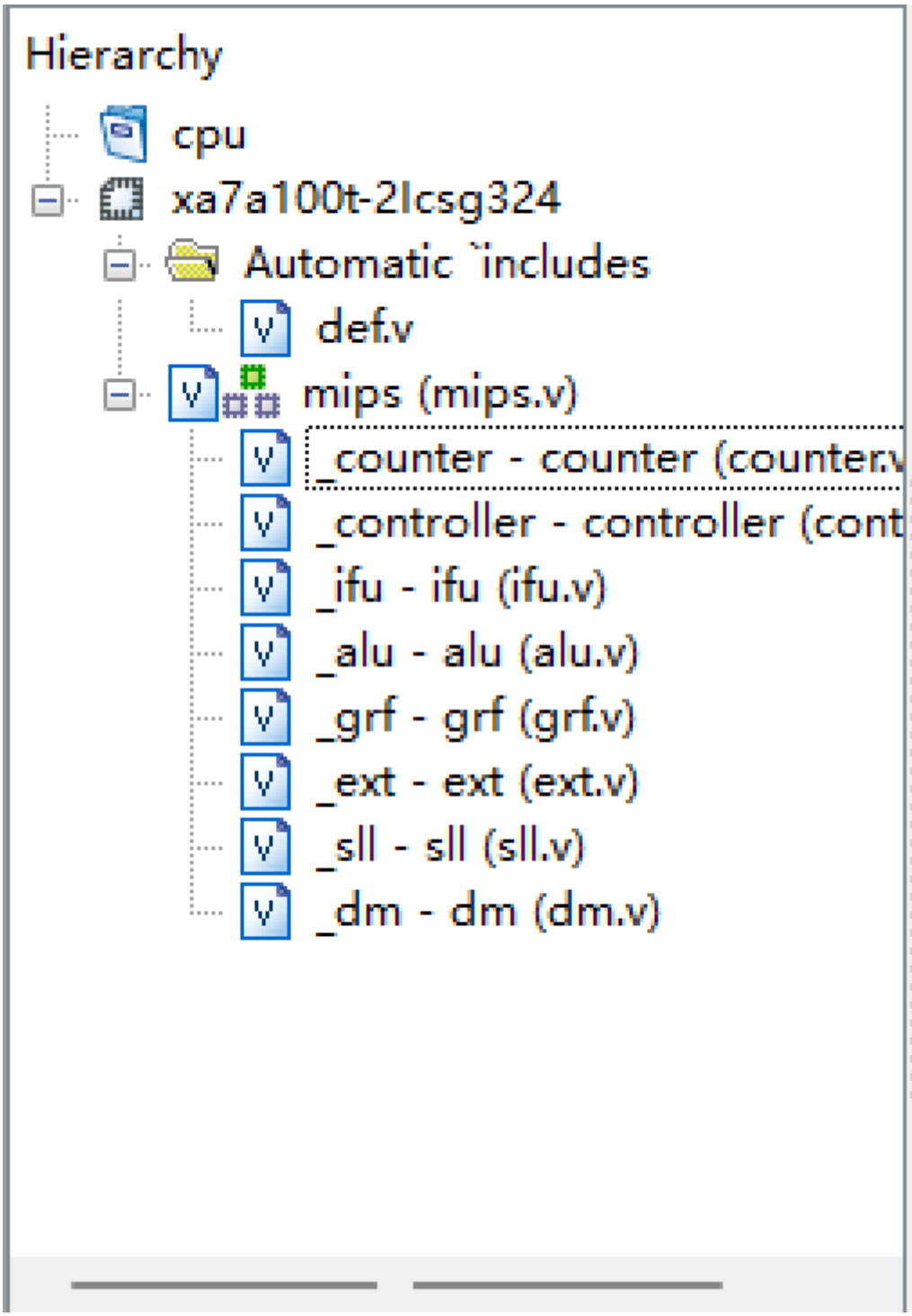
包含对所有参数的宏定义

```
1  `define branchSel_pc4 0
2  `define branchSel_pcOffset 1
3
4  `define jalSel_oldPc 0
5  `define jalSel_jalPc 1
6
7  `define jrSel_oldPc 0
8  `define jrSel_jrPc 1
9
10 `define counter_0 0
11 `define counter_1 1
12
13 `define op_aluOrNop 6'b000000
14 `define op_ori 6'b001101
15 `define op_lw 6'b100011
16 `define op_sw 6'b101011
17 `define op_beq 6'b000100
18 `define op_lui 6'b001111
19 `define op_lb 6'b100000
20 `define op_sb 6'b101000
21 `define op_jal 6'b000011
22
```

```
23 `define funct_add 6'b100000
24 `define funct_sub 6'b100010
25 `define funct_nop 6'b000000
26 `define funct_jr 6'b001000
27
28 `define aluOp_add 0
29 `define aluOp_sub 1
30 `define aluOp_or 2
31
32 `define aluSrc_rd2 0
33 `define aluSrc_signImm 1
34 `define aluSrc_zeroImm 2
35
36 `define regDst_16_20 0
37 `define regDst_11_15 1
38 `define regDst_31 2
39
40 `define sllOp_sign2 0
41 `define sllOp_zero16 1
42
43 `define wdOp_w 0
44 `define wdOp_b 1
45
46 `define memtoReg_aluResult 0
47 `define memtoReg_rdw 1
48 `define memtoReg_sllImm 2
49 `define memtoReg_rdb 3
50 `define memtoReg_pc4
```

H2 mips.v

包含对电路各模块（包括大模块和小模块）的定义和电线的排布



---

counter.v

counter控制初始时pc为0

```
    counter _counter (
        .clk(clk),
        .reset(reset),
        .counter(counter)
        );
```

H2 controller.v

	add	sub	nop	ori	lw	sw	beq	lui	lb	sb	jal	jr
MemtoReg	0	0	\	0	1	\	\	2	3	\	4	\
MemWrite	0	0	0	0	0	1	0	0	0	1	1	0
Branch	0	0	0	0	0	0	1	0	0	0	0	0
ALUControl	0	1	\	2	0	0	1	\	0	0	\	\
ALUSrc	0	0	\	2	1	1	0	\	1	1	\	\
RegDst	1	1	\	0	0	\	\	0	0	\	2	\
RegWrite	1	1	0	1	1	0	0	1	1	0	1	0
SllOp	\	\	\	\	\	\	0	1	\	\	\	\
WDOp	\	\	\	\	\	0	\	\	\	1	\	\
jrSel	0	0	0	0	0	0	0	0	0	0	0	1
jalSel	0	0	0	0	0	0	0	0	0	0	1	0

```
1     controller _controller (
2     .op(op),
3     .funct(funct),
4     .memtoReg(memtoReg),
5     .memWrite(memWrite),
6     .branch(branch),
7     .aluControl(aluControl),
8     .aluSrc(aluSrc),
9     .regDst(regDst),
10    .regWrite(regWrite),
11    .sllOp(sllOp),
12    .wdOp(wdOp),
13    .jalSel(jalSel),
14     .jrSel(jrSel)
15    );
```

---

ifu.v

ifu是最先完成的模块，所以集成度较差

```
ifu _ifu (
.clk(clk),
.reset(reset),
.counter(counter),
.branch(branch),
 .zero(zero),
 .jalSel(jalSel),
 .jrSel(jrSel),
.pc4(pc4),
.pcOffset(pcOffset),
.jalPc(jalPc),
 .jrPc(jrPc),
.pc(pc),
.instr(instr)
);
```

---

grf.v

```
grf _grf (
.a1(a1),
.a2(a2),
.a3(a3),
.wd3(wd3),
 .pc(pc),
 .instr(instr),
.we(regWrite), //we = regWrite
.clk(clk),
.reset(reset),
.rd1(rd1),
.rd2(rd2)
);
```

---

alu.v

```
alu _alu (
.srcA(rd1),
.srcB(srcB),
.aluOp(aluControl),
.zero(zero),
.aluResult(aluResult)
);
```

---

dm.v

```
dm _dm (
.addr(aluResult),
.wdOp(wdOp),
.wd(wd),
 .pc(pc),
.we(memWrite),//we = memWrite
.clk(clk),
.reset(reset),
.rdw(rdw),
.rdb(rdb)
);
```

---

ext.v

```
ext _ext (
.imm(imm),
.signImm(signImm),
.zeroImm(zeroImm)
);
```

---

sll.v

```
    sll _sll (
    .imm(imm),
    .sllOp(sllOp),
    .sllImm(sllImm)
    );
```

其它主电路模块

getPc4

```
    always @(*) begin : getPc4
        pc4 = pc + 4;
    end
```

getJalPc

```
    always @(*) begin : getJalPc
        jalPc = {pc[31:28],instr[25:0],{2{1'b0}}};
    end
```

muxSrcB

```
    always @(*) begin : muxSrcB
        case (aluSrc)
            `aluSrc_rd2 : srcB = rd2;
            `aluSrc_signImm : srcB = signImm;
            `aluSrc_zeroImm : srcB = zeroImm;
            default : srcB = 0;
        endcase
    end
```

H3 muxA3

```
    always @(*) begin : muxA3
        case (regDst)
            `regDst_16_20 : a3 = instr[20:16];
            `regDst_11_15 : a3 = instr[15:11];
            `regDst_31 : a3 = 31;
            default : a3 = 0;
        endcase
    end
```

H3 getPcOffset

```
    always @(*) begin : getPcOffset
        pcOffset <= sllImm + pc4;
    end
```

H3 muxGetWd3

```
    always @(*) begin : muxGetWd3
        case (memtoReg)
            `memtoReg_aluResult : wd3 = aluResult;
            `memtoReg_rdw : wd3 = rdw;
            `memtoReg_sllImm : wd3 = sllImm;
            `memtoReg_rdb : wd3 = rdb;
            `memtoReg_pc4 : wd3 = pc4;
            default : wd3 = 0;
        endcase
    end
```

H2 其它主电路assign连接

```
    assign op = instr[31:26];
    assign funct = instr [5:0];
    assign imm = instr[15:0];
    assign wd = rd2;
    assign jrPc = rd1;
    assign a1 = instr[25:21];
    assign a2 = instr[20:16];
    assign wd = rd2;
```

H2 mips定义电线汇总

```
    wire counter;//
    wire [31:0] instr;//
    wire [5:0] op;//
    wire [5:0] funct;//
    wire [2:0] memtoReg;//
    wire memWrite;//
    wire branch;//
    wire [1:0] aluControl;//
    wire [1:0] aluSrc;//
    wire [1:0] regDst;//
    wire regWrite;//
    wire sllOp;//
    wire wdOp;//
    wire jalSel;//
    wire jrSel;
    reg [31:0] pc4;//
    reg [31:0] jalPc;//
    wire [31:0] jrPc;
    wire zero;//
    wire [31:0] pc;//
    reg [31:0] pcOffset;//
    reg [31:0] srcB;//
```

```
23      wire [31:0] rd1;//
24      wire [31:0] rd2;//
25      wire [31:0] signImm;//
26      wire [31:0] zeroImm;//
27      wire [31:0] aluResult;//
28      wire [4:0] a1;//
29      wire [4:0] a2;//
30      reg [4:0] a3;//
31      reg [31:0] wd3;//
32      wire [15:0] imm;//
33      wire [13:0] a;//
34      wire [31:0] sllImm;
35      wire [31:0] wd;
36      wire [31:0] rdw;//
37      wire [7:0] rdb;//
```

---

1. 阅读下面给出的 DM 的输入示例中（示例 DM 容量为 4KB，即 32bit × 1024字），根据你的理解回答，这个 addr 信号又是从哪里来的？地址信号 addr 位数为什么是 [11:2] 而不是 [9:0]？

文件	模块接口定义

dm.v	dm(clk,reset,MemWrite,addr,din,dout);
	input clk; //clock
	input reset; //reset
	input MemWrite; //memory write enable
	input [11:2] addr; //memory's address for write
	input [31:0] din; //write data
	output [31:0] dout; //read data

   答：addr是从alu中计算得来的；由于dm为按字编址，所以在访问时，要将addr的低两位舍去，以达到乘四的效果，使dm访问正确的地址（不考虑地址不匹配的情况）

2. 思考上述两种控制器设计的译码方式，给出代码示例，并尝试对比各方式的优劣。

   答：这两种控制器设计的译码方式为"**指令对应的控制信号如何取值**"和"**控制信号每种取值所对应的指令**"；

```
//指令对应的控制信号如何取值
if (add) begin
    regDst = 1;
    regWrite = 1;
end
```

```
//控制信号每种取值所对应的指令
memWrite = sw | sb | jal;
```

译码方式	优点	缺点
指令对应的控制信号如何取值	新添指令比较形象	编写比较困难，不容易调试
控制信号每种取值所对应的指令	编写简单，调试方便	新添指令不够形象

3. 在相应的部件中，复位信号的设计都是**同步复位**，这与 P3 中的设计要求不同。请对比**同步复位**与**异步复位**这两种方式的 reset 信号与 clk 信号优先级的关系。

   答：同步复位中，clk信号优先级高于reset；异步复位中，reset信号优先级高于clk

4. C 语言是一种弱类型程序设计语言。C 语言中不对计算结果溢出进行处理，这意味着 C 语言要求程序员必须很清楚计算结果是否会导致溢出。因此，如果仅仅支持 C 语言，MIPS 指令的所有计算指令均可以忽略溢出。 请说明为什么在忽略溢出的前提下，addi 与 addiu 是等价的，add 与 addu 是等价的。提示：阅读《MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set》中相关指令的 Operation 部分。

   根据 RTL 语言描述： addi 与 addiu 的区别在于当出现溢出时，addiu 忽略溢出，并将溢出的最高位舍弃； addi 会报告 SignalException(IntegerOverflow) 故忽略溢出，二者等价。