多周期处理器技术报告

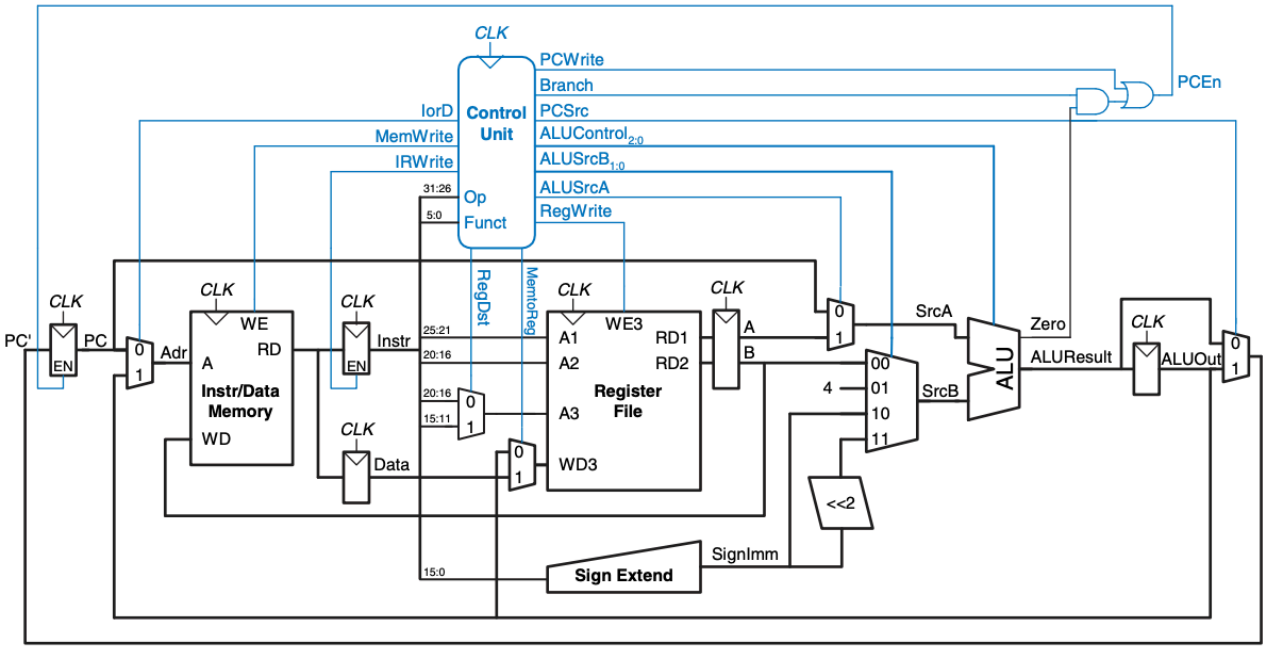
王艺坤 20 CS

1.Introduction

32 位 MIPS 多周期处理器，由 system verilog[1] 编写。

> A **multi-cycle processor** is a processor that carries out one instruction over multiple clock cycles, often without starting up a new instruction in that time (as opposed to a pipelined processor).[2]

> 为什么需要多周期处理器?
>
> 1. 由于木桶效应，单周期处理器需要足够长的周期来完成最慢的指令。多周期将指令的执行分成多个较短的周期，以此尝试减少整体的执行时间：某些指令只需要更少的周期数，比如 `jne` 可以在 3 个周期内完成指令的执行。
> 2. 单周期处理器需要 3 个加法器 （ALU 中一个和 PC 更新中的两个），多周期可以复用 ALU 中的加法器来减小电路规模，从而降低成本。



1.1 已实现指令集

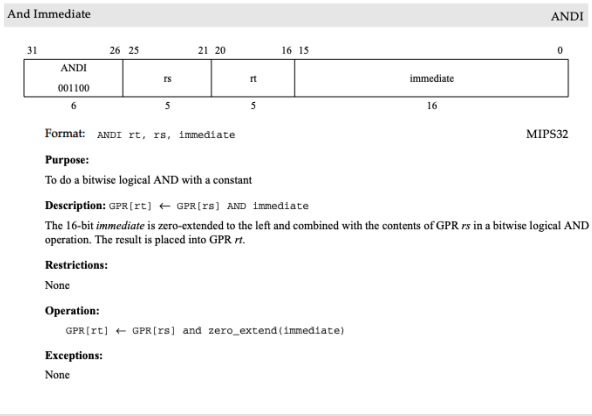
按照 《数字设计和计算机体系结构》[^ 1]中单周期 CPU 设计，处理器应当包括一下 11 种指令：

- 5种 RTYPE 运算指令：`add, sub, and, or, slt`
- 1种 IMM 运算指令：`addi`
- 2种IO指令：`lw, sw`
- 1种跳转指令：`j`
- 1种分支指令：`beq`
- bubble：`nop`

考虑到提升处理器的拓展性，额外加入三条 IMM 运算指令与 1 种跳转指令。

新加入的指令包括 `ori`, `andi`, `slti`, `bne`：

ORI



And Immediate ANDI

31–26	25–21	20–16	15–0
ANDI 001100	rs	rt	immediate
6	5	5	16

Format: `ANDI rt, rs, immediate` MIPS32

Purpose:

To do a bitwise logical AND with a constant

Description: `GPR[rt] ← GPR[rs] AND immediate`

The 16-bit *immediate* is zero-extended to the left and combined with the contents of GPR *rs* in a bitwise logical AND operation. The result is placed into GPR *rt*.

Restrictions:

None

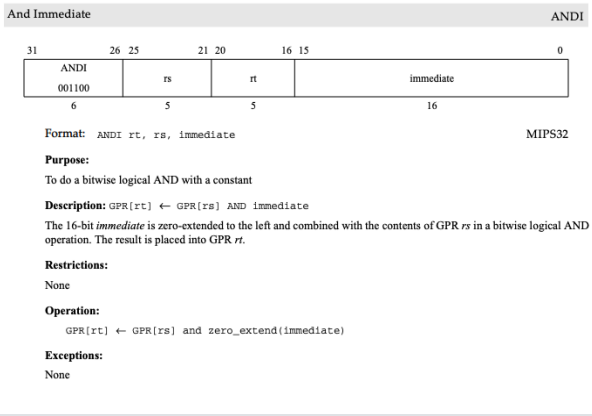
Operation:

```
GPR[rt] ← GPR[rs] and zero_extend(immediate)
```

Exceptions:

None

ANDI



And Immediate ANDI

31–26	25–21	20–16	15–0
ANDI 001100	rs	rt	immediate
6	5	5	16

Format: `ANDI rt, rs, immediate` MIPS32

Purpose:

To do a bitwise logical AND with a constant

Description: `GPR[rt] ← GPR[rs] AND immediate`

The 16-bit *immediate* is zero-extended to the left and combined with the contents of GPR *rs* in a bitwise logical AND operation. The result is placed into GPR *rt*.

Restrictions:

None

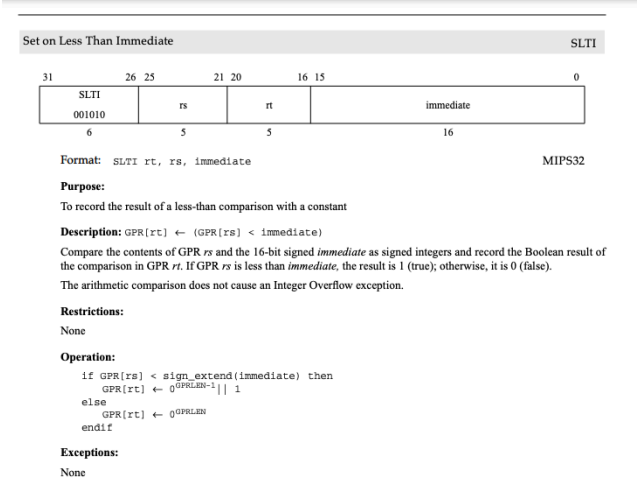
Operation:

```
GPR[rt] ← GPR[rs] and zero_extend(immediate)
```

Exceptions:

None

SLTI



Set on Less Than Immediate SLTI

31–26	25–21	20–16	15–0
SLTI 001010	rs	rt	immediate
6	5	5	16

Format: `SLTI rt, rs, immediate` MIPS32

Purpose:

To record the result of a less-than comparison with a constant

Description: `GPR[rt] ← (GPR[rs] < immediate)`

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers and record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

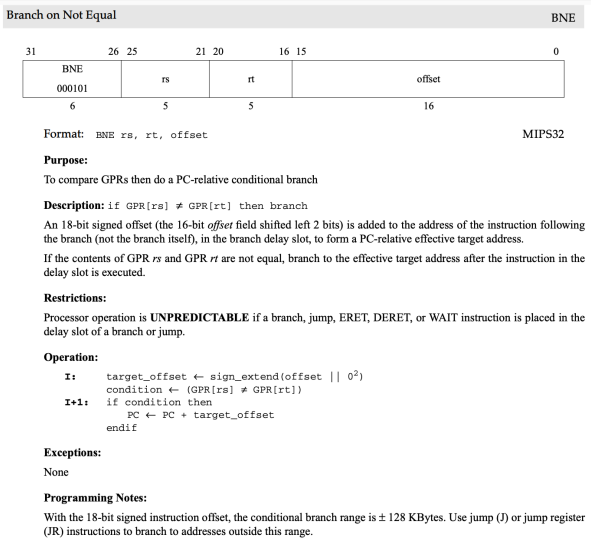
Operation:

```
if GPR[rs] < sign_extend(immediate) then
   GPR[rt] ← 0^(GPRLEN-1) || 1
else
   GPR[rt] ← 0^GPRLEN
endif
```

Exceptions:

None

BNE



Branch on Not Equal BNE

31–26	25–21	20–16	15–0
BNE 000101	rs	rt	offset
6	5	5	16

Format: `BNE rs, rt, offset` MIPS32

Purpose:

To compare GPRs then do a PC-relative conditional branch

Description: `if GPR[rs] ≠ GPR[rt] then branch`

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* and GPR *rt* are not equal, branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

```
I:    target_offset ← sign_extend(offset || 0^2)
      condition ← (GPR[rs] ≠ GPR[rt])
I+1:  if condition then
         PC ← PC + target_offset
      endif
```

Exceptions:

None

Programming Notes:

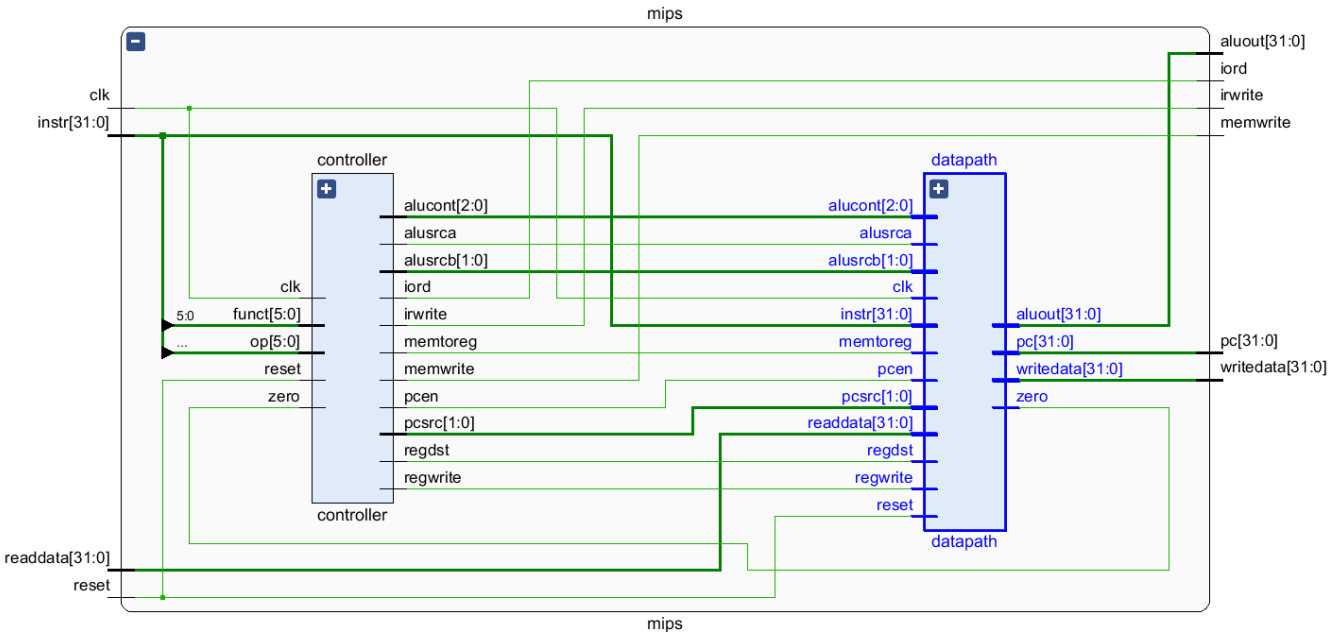
With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

overall instruction set:

```
add     $rd, $rs, $rt          # [rd] = [rs] + [rt]
sub     $rd, $rs, $rt          # [rd] = [rs] - [rt]
and     $rd, $rs, $rt          # [rd] = [rs] & [rt]
or      $rd, $rs, $rt          # [rd] = [rs] | [rt]
slt     $rd, $rs, $rt          # [rd] = [rs] < [rt] ? 1 : 0
addi    $rt, $rs, imm          # [rt] = [rs] + SignImm
andi    $rt, $rs, imm          # [rt] = [rs] & ZeroImm
ori     $rt, $rs, imm          # [rt] = [rs] | ZeroImm
slti    $rt, $rs, imm          # [rt] = [rs] < SignImm ? 1 : 0
lw      $rt, imm($rs)          # [rt] = [Address]
sw      $rt, imm($rs)          # [Address] = [rt]
j       label                  # PC = JTA
beq     $rs, $rt, label        # if ([rs] == [rt]) PC = BTA
bne     $rs, $rt, label        # if ([rs] != [rt]) PC = BTA
nop                            # No operation
```

2. 实现

多周期处理器与单周期处理器在实现上的异同：



1. 总体上都保持了 controller - datapath 两个主要模块的架构。controller 包括一个 main decoder 以及一个 ALU decoder，通过 `ALUcont` 信号控制 ALU 行为，此外通过 `pcen` 信号控制处理器跳转。datapath 确定 pc 更新行为，包含 ALU 以及 register file。
2. 数据路径上，将体系结构状态元件和储存器与组合逻辑连接来创建一条数据路径。增加了非体系结构状态元件来保存每个步骤之间的中间结果。
3. 每个指令执行期间控制器针对不同的步骤上产生不同的指令，因此控制器是有限状态机而非组合电路。

2.1 controller

与单周期处理器一样， controller 通过指令的 op 字段和 funct 字段计算控制信号。第一张图给出了一个完整多周期 MIPS 处理器的控制单元和数据路径的连接。

与单周期处理器一样，控制单元分为一个主译码器和一个 ALU 译码器。

其中 ALU 译码器依然保持着单周期处理器的真值表：

op	ALUcont
0'b000 (ALU_AND)	0'b000
0'b001 (ALU_OR)	0'b001
0'b010 (ALU_ADD)	0'b010
0'b011 (ALU_NO_USE)	depend on instr[5:0] (`funct`)
0'b110 (ALU_SUB)	0'b110
0'b111 (ALU_SLT)	0'b111

而由于指令所耗周期数依赖于译码结果，主译码器被设计为一个大状态机：

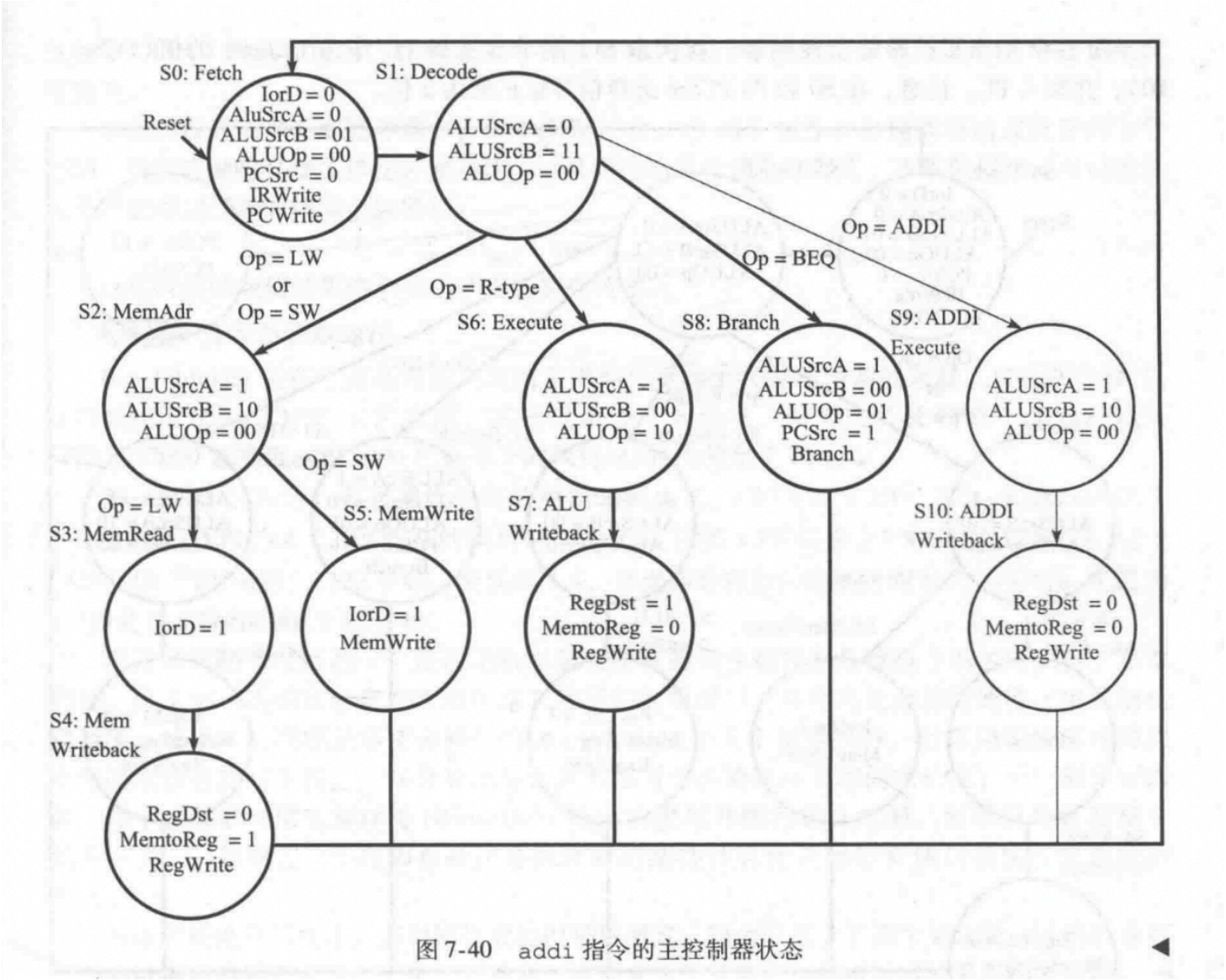


图 7-40　addi 指令的主控制器状态

指令的执行可以认为是一个状态转移的过程。

```
// multicircle/controller.sv
// author: ekon wang

always_comb begin
  unique case(state)
    FETCH:      nextstate = DECODE;
    DECODE: case(op)
      LW:         nextstate = MEMADR;
      SW:         nextstate = MEMADR;
      // branch.
      BEQ:        nextstate = BEQEX;
      BNE:        nextstate = BNEEX;
      // imm.
      ADDI:       nextstate = ADDIEX;
      ANDI:       nextstate = ANDIEX;
      ORI:        nextstate = ORIEX;
      SLTI:       nextstate = SLTIEX;
      // jump.
      J:          nextstate = JEX;
```

```
      RTYPE: case(funct)
        BUBBLE:     nextstate = FETCH;
        default:    nextstate = RTYPEEX;
      endcase
      default:    nextstate = 5'bx;
    endcase
    MEMADR: case(op)
      LW:         nextstate = MEMRD;
      SW:         nextstate = MEMWR;
      default:    nextstate = 5'bx;
    endcase
    MEMRD:      nextstate = MEMWB;
    MEMWB:      nextstate = FETCH;
    MEMWR:      nextstate = FETCH;
    RTYPEEX:    nextstate = RTYPEWB;
    RTYPEWB:    nextstate = FETCH;
    // branch.
    BEQEX:      nextstate = FETCH;
    BNEEX:      nextstate = FETCH;
    // imm.
    ADDIEX:     nextstate = ADDIWB;
    ANDIEX:     nextstate = ANDIWB;
    ORIEX:      nextstate = ORIWB;
    SLTIEX:     nextstate = SLTIWB;
    ADDIWB:     nextstate = FETCH;
    ANDIWB:     nextstate = FETCH;
    ORIWB:      nextstate = FETCH;
    SLTIWB:     nextstate = FETCH;
    // jump.
    JEX:        nextstate = FETCH;
    default:    nextstate = 5'bx;
  endcase
end

always_comb begin
  unique case(state)
    FETCH:      controls = {FETCH_C,    ALU_ADD};
    DECODE:     controls = {DECODE_C,   ALU_ADD};
    MEMADR:     controls = {MEMADR_C,   ALU_ADD};
    MEMRD:      controls = {MEMRD_C,    ALU_ADD};
    MEMWB:      controls = {MEMWB_C,    ALU_ADD};
    MEMWR:      controls = {MEMWR_C,    ALU_ADD};
    RTYPEEX:    controls = {RTYPEEX_C,  ALU_NO_USE};
    RTYPEWB:    controls = {RTYPEWB_C,  ALU_ADD};
    // branch instructions.
    BEQEX:      controls = {BEQEX_C,    ALU_SUB};
    BNEEX:      controls = {BNEEX_C,    ALU_SUB};

    // imm instructions.
```

```
    ADDIEX:     controls = {ADDIEX_C,   ALU_ADD};
    ADDIWB:     controls = {ADDIWB_C,   ALU_ADD};
    ANDIEX:     controls = {ADDIEX_C,   ALU_AND};
    ANDIWB:     controls = {ADDIWB_C,   ALU_ADD};
    ORIEX:      controls = {ADDIEX_C,   ALU_OR};
    ORIWB:      controls = {ADDIWB_C,   ALU_ADD};
    SLTIEX:     controls = {ADDIEX_C,   ALU_SLT};
    SLTIWB:     controls = {ADDIWB_C,   ALU_ADD};

    JEX:        controls = {JEX_C,      ALU_ADD};
    default:    controls = 16'hxxxx;
  endcase
end
```

在实现中，为了 code 的可读性，我尝试利用 system verilog 的 parameter 语法，定义了大量全局变量。这些定义我统一放在了 `common.svh` 。

```
// common.svh
// author: ekon wang

// @ALU
parameter ALU_AND       =       3'b000;
parameter ALU_OR        =       3'b001;
parameter ALU_ADD       =       3'b010;
parameter ALU_NO_USE    =       3'b011;
parameter ALU_RAND      =       3'b100;
parameter ALU_ROR       =       3'b101;
parameter ALU_SUB       =       3'b110;
parameter ALU_SLT       =       3'b111;

// ALUDEC
parameter FUNCT_ADD     =       6'b100000;
parameter FUNCT_SUB     =       6'b100010;
parameter FUNCT_AND     =       6'b100100;
parameter FUNCT_OR      =       6'b100101;
parameter FUNCT_SLT     =       6'b101010;

parameter RTYPE     =       6'b000000;
parameter LW        =       6'b100011;
parameter SW        =       6'b101011;
parameter BEQ       =       6'b000100;
parameter ADDI      =       6'b001000;
parameter J         =       6'b000010;
parameter BUBBLE    =       6'b000000;
parameter BNE       =       6'b000101;
parameter ORI       =       6'b001101;
parameter ANDI      =       6'b001100;
parameter SLTI      =       6'b001010;
```

```
// FSM defines
parameter FETCH     =       5'b00000;
parameter DECODE    =       5'b00001;
parameter MEMADR    =       5'b00010;
parameter MEMRD     =       5'b00011;
parameter MEMWB     =       5'b00100;
parameter MEMWR     =       5'b00101;
parameter RTYPEEX   =       5'b00110;
parameter RTYPEWB   =       5'b00111;
parameter BEQEX     =       5'b01000;
parameter ADDIEX    =       5'b01001;
parameter ADDIWB    =       5'b01010;
parameter JEX       =       5'b01011;
parameter BNEEX     =       5'b01100;
parameter ORIEX     =       5'b01101;
parameter ORIWB     =       5'b01110;
parameter ANDIEX    =       5'b01111;
parameter ANDIWB    =       5'b10000;
parameter SLTIEX    =       5'b10001;
parameter SLTIWB    =       5'b10010;

// FSM control code
parameter FETCH_C       =       14'h1404;
parameter DECODE_C      =       14'h000c;
parameter MEMADR_C      =       14'h0108;
parameter MEMRD_C       =       14'h0040;
parameter MEMWB_C       =       14'h0220;
parameter MEMWR_C       =       14'h0840;
parameter RTYPEEX_C     =       14'h0100;
parameter RTYPEWB_C     =       14'h0210;
parameter BEQEX_C       =       14'h0181;
parameter BNEEX_C       =       14'h2101;
parameter ADDIEX_C      =       14'h0108;
parameter ADDIWB_C      =       14'h0200;
parameter JEX_C         =       14'h1002;
```

2.2 datapath

2.2.1 pc 更新

按照有限状态机设计，pc 在 fetch 阶段之后即更新为 pc + 4。对于跳转指令和分支指令，`pcen` 信号将在执行阶段使能，将 pc 更新为 pcnext。

```
assign pcen = (beq & zero) | (bne & !zero) | pcwrite;
```

其中 pcnext 复用器的逻辑如下：

```
always_comb begin
  unique case(pcsrc)
    2'b00:  pcnext = aluresult;
    2'b01:  pcnext = aluout;
    2'b10:  pcnext = {pc[31:28], signimmsh[27:0]};
    default:
      pcnext = 'x;
  endcase
end
```

三个分支分别对应 fetch 阶段 pc = pc + 4，分支指令以及跳转指令时的更新逻辑。

2.2.2 regfile

regfile 被实现为一个简单的存储器结构，当索引 index 为 0 时，结果固定为 0。

```
module regfile(
    input   u1      clk,
    input   u1      we3,
    input   u5      ra1, ra2, wa3,
    input   u32     wd3,
    output  u32     rd1, rd2
);
    u32 registers[31:0];

    // three ported register file
    // note: for pipelined CPU, write third
    // port on falling edge of clk.
    always @(posedge clk) begin
        if (we3) registers[wa3] <= wd3;
    end

    assign rd1 = (ra1 != 0) ? registers[ra1] : '0;
    assign rd2 = (ra2 != 0) ? registers[ra2] : '0;

endmodule
```

2.2.3 ALU

为了保证 ALU 的可拓展性，使用了 system verilog 提供的 parameter 机制，以便将来支持大的位宽，比如 64 位。

此外 ALU 实际除了指令集中的 `and`, `add`, `sub`, `slt` 等运算，还支持了 `rand`, `ror` 运算。ALU 的输出除了运算结果，还包含了一个 zero 信号，zero 信号是跳转指令中不可或缺的。

```
module ALU #(
    N = 32
) (
    input  logic  [N-1:0]   A,
```

```
    input  logic  [N-1:0]   B,
    input  logic  [2:0]     alucont,
    output logic  [N-1:0]   result,
    output logic            zero
);
always_comb begin
    unique case(alucont)
        ALU_AND     : result = A & B;
        ALU_OR      : result = A | B;
        ALU_ADD     : result = A + B;

        ALU_RAND    : result = A & ~B;
        ALU_ROR     : result = A | ~B;
        ALU_SUB     : result = A - B;
        ALU_SLT     : result = A < B ? 1 : 0;
        default     : result = 'x;
    endcase
end
assign zero = !result;

endmodule
```

2.3 driver

为了更好的上板，在 IO 部分尝试编写了一个小型驱动，方便之后上板适用。

代码包括一个约束文件以及两个模块 `map` （负责 dmem 的读入与总体 IO）以及 `mux7seg` （负责输出信号到 7 段数码管的映射）。

3. 测试

3.1 单元测试

尝试更加精细化的调试，因此给了主要指令提供了一系列的仿真单元测试。

指令	测试文件
beq	benchtest/branch.sv
nop	benchtest/bubble.sv
addi	benchtest/imm.sv
jmp	benchtest/jmp.sv
lw	benchtest/lw.sv
sw	benchtest/sw.sv
add, and, sub	benchtest/rtype.sv
ISA-1	benchtest/seq.sv
ISA-extended	benchtest/seq-ext.sv

其中 ISA-1 指《数字设计和计算机体系结构》[^ 1]中多周期 CPU 设计，处理器应当包括的 11 种指令。

ISA-extended 指在 11 种指令基础上，经过拓展的指令集。

seq-ext.sv 最长，且包含所有 15 类指令，由于篇幅限制，只展示 seq-ext.sv 测试过程及结果：

当最后处理器尝试在 0d88 处写入 value 30 时，即测试通过。

```
always begin
        #1;
        if (memwrite & (dataaddr != 84 & dataaddr != 80 & dataaddr != 88)) begin
            $display("Simulation failed");
            $stop;
        end else if (memwrite & dataaddr == 80) begin
            if (writedata == 7) begin
                $display("Milestone hit");
            end else begin
                $display("Simulation failed");
                $stop;
            end
        end else if (memwrite & dataaddr == 84) begin
            if (writedata == 7) begin
                $display("Milestone hit");
            end else begin
                $display("Simulation failed");
                $stop;
            end
        end else if (memwrite & dataaddr == 88) begin
            if (writedata == 30) begin
                $display("Succeeded");
                $stop;
            end else begin
```

```
                $display("Simulation failed");
                $stop;
            end
        end
    end
```

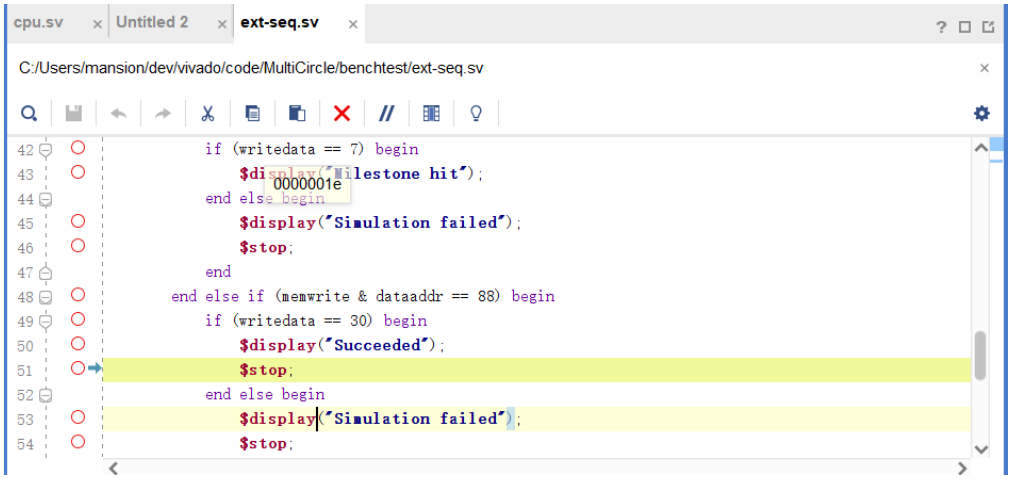
打开 vivado 项目，在 `cpu` 顶层模块之上增加 benchtest 模块（直接添加仿真文件 `seq-ext.sv` 即可），将 mem 模块中读入的指令，改为 `ext-seq.dat` 确保内存指令无误，点击运行 run simulation，开始仿真。

终端输出：

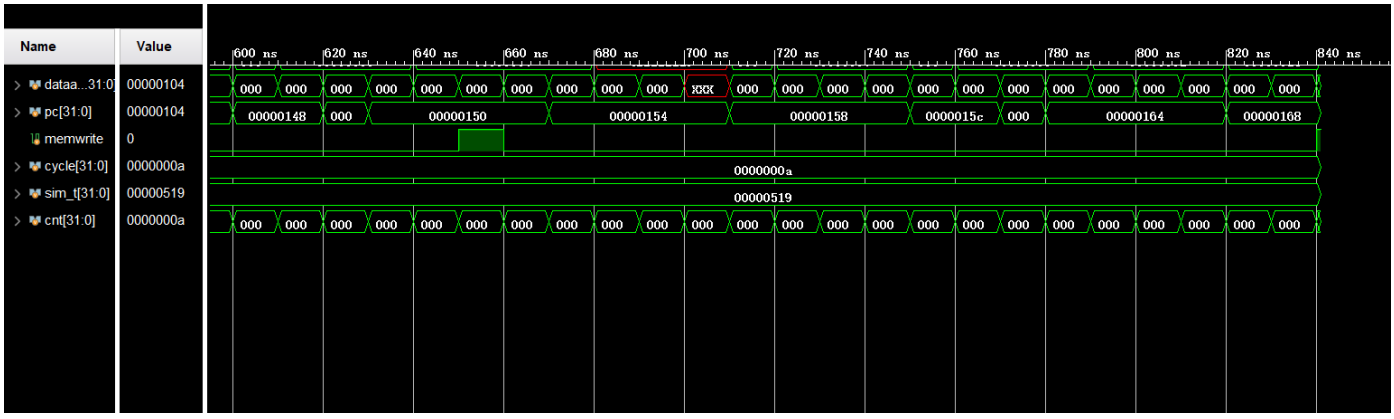
```
epoch           83


epoch           84
Succeeded
```

测试终止断点：



波形输出：



测试通过。

4.Contributor

- Ekon Wang - Fudan University

5. License

6. Reference

1. https://en.wikipedia.org/wiki/SystemVerilog ↩

2. https://en.wikipedia.org/wiki/Multi-cycle_processor ↩