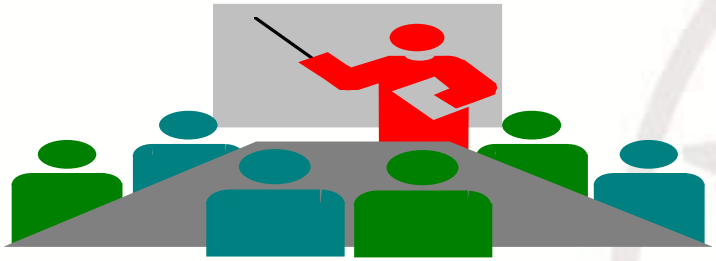


计算机组成与设计

Computer Organization & Design

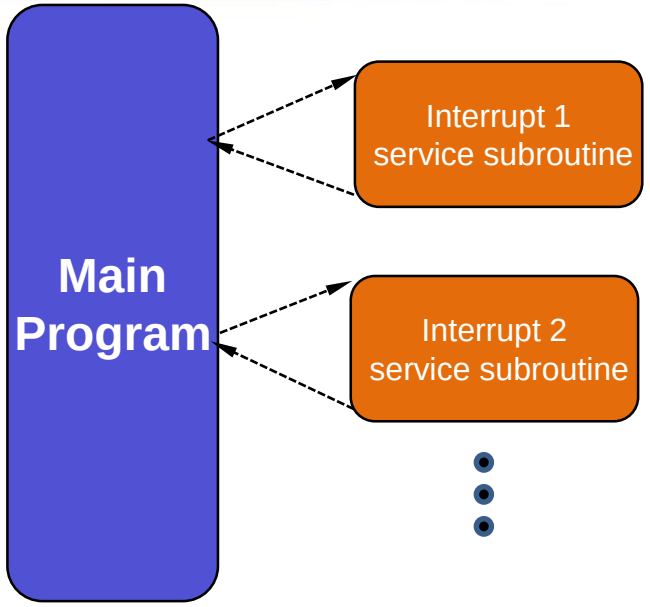
The Hardware/Software Interface

Chapter 4

Processor Within Interruption and Exception

施青松
Asso. Prof. Shi Qingsong
College of Computer Science and Technology, Zhejiang University
zjsqs@zju.edu.cn

Processor Design…



CPU within Exception

4.9 Exception (Interruption)

- **The cause of changing CPU's work flow :**
  - Control instructions in program (bne/beq, j/ jal , etc)

    It is foreseeable in programming flow
  - Something happen suddenly (Exception and Interruption)

    It is unpredictable
    - Call Instructions triggered by hardware
- **Unexpected events**
  - Exception: from within processor (overflow, undefined instruction, etc)
  - Interruption : from outside processor (input /output)

Exception

- **Exception**

  An Exception is a unexpected event from within processor.

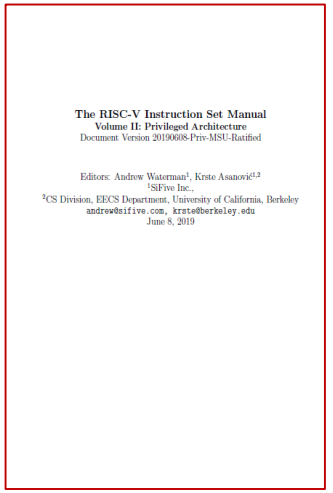
- **We follow the RISC convention**
  - using the term exception to refer to any unexpected change in control flow
- **Here we will discuss two types exceptions :**
  - undefined instruction、arithmetic overflow
  - Interruption by IO、System call
- **Interruption**
  - Implementation methods are similar

Improve CPU Running efficiency

- **A method of interacting between CPU and Peripherals**
  - Polling
    - The processor periodically checks status bit
  - **Interrupts/Exceptions**
    - It causes processor to be interrupted, when the I/O device needs to notify the processor.
  - DMA
    - CPU don't participate in the interactive
    - Big data transmission
- **Interrupt property**
  - as **jal**/**call** instructions by Hardware-triggered
    - between instruction or within instruction
  - Soft interrupt
    - jal instruction, is implemented through hardware interrupt mechanism

How Exceptions Are Handled

- **What must do the processor?**
  - When exception happens the processor must do something
  - The predefined process routines are saved in memory when computer starts
- **Problem**
  - how can CPU go to relative routine when an exception occurs
  - CPU should know
    - The cause of exception
    - which instruction generate the exception
  - Like the “jal" instruction, but the hardware triggers


The RISC-V Instruction Set Manual
Volume II: Privileged Architecture
Document Version 20190608-Priv-MSU-Ratified

Editors: Andrew Waterman[1], Krste Asanović[1,2]
[1]SiFive Inc.,
[2]CS Division, EECS Department, University of California, Berkeley
andrew@sifive.com, krste@berkeley.edu
June 8, 2019

RISC-V Privileged

All hardware implementations must provide M-mode

RISC-V Privileged Architecture

- The machine level has the highest privileges
  - and is the only mandatory privilege level for a RISC-V hardware platform.
  - Code run in machine-mode (M-mode) is usually inherently trusted, as it has low-level access to the machine implementation.
  - M-mode can be used to manage secure execution environments on RISC-V.
- User-mode (U-mode) and supervisor-mode (S-mode) are intended for conventional application and operating system usage respectively.

Level	Encoding	Name	Abbreviation	
0	00	User/Application	U	用户模式
1	01	Supervisor	S	监督模式
2	10	Reserved		保留
3	11	Machine	M	机器模式

RISC-V Privilege Modes Usage

- **Each privilege level has**
  - a core set of privileged ISA extensions
    - with optional extensions and variants.
- **Combinations of privilege modes**
  - Implementations might provide anywhere from 1 to 3 privilege modes
    - trading off reduced isolation for lower implementation cost

Number of levels	Supported Modes	Intended Usage
1	M	Simple embedded systems
2	M, U	Secure embedded systems
3	M, S, U	Systems running Unix-like operating systems

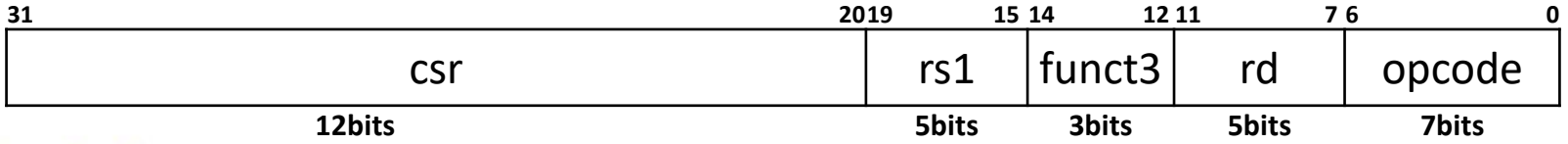
  - For example, machine-mode supports an optional standard extension for memory protection.

RISC-V interrupt structure

- **All hardware implementations must provide M-mode**
  - as this is the only mode that has unfettered access to the whole machine.
  - The simplest RISC-V implementations may provide only M-mode
    - though this will provide no protection against incorrect or malicious application code
- **Machine mode(M-mode) most important task**
  - that is to intercept and handle interrupts/exceptions
  - There are 4096 Control and Status Registers (CSRs)
    - Similar to CP0 of MIPS

Control and Status Registers (CSRs)

- **CSRs, are additional set of regisiters**
  - accessible by some subset of the privilege levels using the CSR instructions
  - These can be divided into two main classes:
    - those that atomically read-modify-write control and status registers
    - and all other privileged instructions.
- **All privileged instructions encode by the SYSTEM major opcode**
  - CSRs and instructions are associated with one privilege level, they are also accessible at all higher privilege levels
    - The standard RISC-V ISA sets aside a 12-bit encoding space (csr[11:0]) for up to 4,096 CSRs.



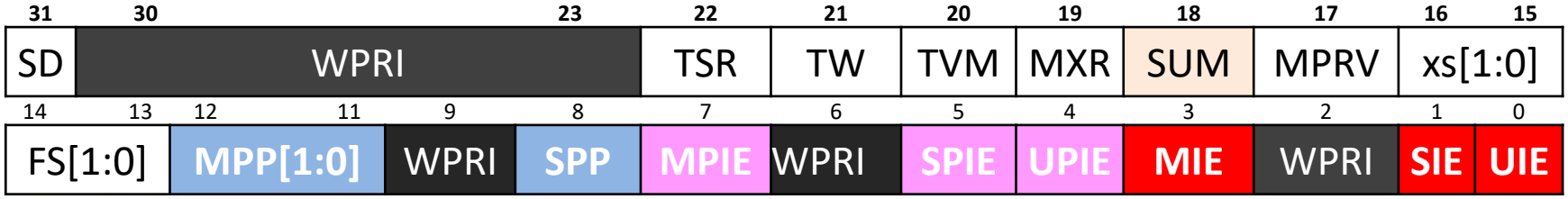
Exception & Interrupt related registers

CSR	Privilege	Abbr.	Name	Description
0x300	MRW	mstatus	Machine STATUS register 机器模式状态寄存器	MIE、MPIE域标记中断全局使能
0x304	MRW	mie	Machine Interrupt Enable register 机器模式中断使能寄存器	控制不同类型中断的局部使能
0x305	MRW	mtvec	Machine trap-handler base address 机器模式异常入口基地址寄存器	进入异常服务程序基地址
0x341	MRW	mepc	Machine exception program counter 机器模式异常PC寄存器	异常断点PC地址
0x342	MRW	mcause	Machine trap cause register 机器模式原因寄存器	处理器异常原因
0x343	MRW	mtval	Machine Trap Value register 机器模式异常值寄存器	处理器异常值地址或指令
0x344	MRW	mip	Machine interrupt pending 机器模式中断挂起寄存器	处理器中断等待处理

Interrupt Registers： mstatus(*0x300*)

- Machine STATUS register
  - These bits are primarily used to guarantee atomicity with respect to interrupt handlers in the current privilege mode.

bit	Name	Attributes	Description
0	UIE	RW	User interrupt enable
1	SIE	RW	Supervisor interrupt enable
3	MIE	RW	Machine interrupt enable
4	UPIE	RW	previous user-level interrupts enable
5	SPIE	RW	Previous Supervisor interrupt-enable
7	MPIE	RW	Previous Machine interrupt enable
8	SPP	RW	Supervisor Previous Privilege mode
12:11	MPP	RW	Machine Previous Privilege mode
……			
31:23,9,6,2	WPRI		Reserved



Interrupt Registers： mie/mip(*0x304/344*)

- **Machine Interrupt Enable register**
  - MIE register controls whether it can respond to interrupts, corresponding different modes
    - MEIE、SEIE and UEIE enable external interrupt
    - MSIE、SSIE & USIE enable software interrupts
    - MTIE、STIE and UTIE enable timer interrupts
- **Machine interrupt-pending register**
  - The mip register is an MXLEN-bit read/write register containing information on pending interrupts

	31 … 12	11	10	9	8	7	6	5	4	3	2	1	0
mie	WPRI	MEIE	WPRI	SEIE	UEIE	MTIE	WPRI	STIE	UTIE	MSIE	WPRI	SSIE	USIE
mip	WPRI	MEIP	WPRI	SEIP	UEIP	MTIP	WPRI	STIP	UTIP	MSIP	WPRI	SSIP	USIP

M, S and U correspond to M mode, S mode and U mode interrupt respectively

Interrupt Registers： mtvec (*0x305*)

Machine Trap-Vector Base-Address Register

- The mtvec register holds trap vector configuration, consisting of a vector base address (BASE) and a vector mode (MODE)

- The value in the BASE field must always be aligned on a 4-byte boundary, and the MODE setting may impose additional alignment constraints on the value in the BASE field.

MODE Value	Name	Description
0	Direct(查询)	All exceptions set pc to BASE
1	Vectored(向量)	Asynchronous interrupts set pc to BASE+4×cause
≥2	--	Reserved

BASE+ 4×Exception Code

Interrupt Registers： mepc (*0x341*)

Machine Exception Program Counter

- mepc is a WARL register that must be able to hold all valid physical and virtual addresses.
  - When a trap is taken into M-mode, mepc is written with the address of the instruction that was interrupted or exception.

- The low bit of mepc (mepc[0]) is always zero.
  - On implementations that support only IALIGN=32, the two low bits (mepc[1:0]) are always zero.
- Exception： `mepc ← pc`
- Interrupted： `mepc ← pc + 4`

Interrupt Registers： mcause (*0x342*)

Machine Cause Register (mcause)

- When a trap is taken mcause register is written with a code indicating the event that caused the Exception/Interruption.

- Exception Code与mtvec的向量模式相对应
  - 在异步中断时，不同的模式会跳转到不同的入口
- The Exception Code is a WLRL
  - Write/Read Only Legal Values
  - 如果写入值不合法可以引发非法指令异常

Exception Code mcause (*0x342*)

INT	E Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	Reserved for future standard use
1	**3**	**Machine software interrupt**
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	Reserved for future standard use
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	Reserved for future standard use
1	**11**	**Machine external interrupt**
1	12-15	Reserved for future standard use
1	≥16	Reserved for platform use
0	16-23	Reserved for future standard use
0	32-47	Reserved for future standard use
0	14/≥64	Reserved for future standard use

INT	E Code	Description
0	0	Instruction address misaligned
0	1	Instruction access fault
0	**2**	**Illegal instruction**
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	Reserved
0	11	**Environment call from M-mode**
0	12	Instruction page fault
0	13	Load page fault
0	15	Store/AMO page fault
0	24-31	Reserved for custom use
0	48-63	Reserved for custom use

RISV-V Interrupt priority

External interrupt > Software interrupt > Timer interrupt

Synchronous exception priority

Priority	Exception Code	Description
Highest	3	Instruction address breakpoint
	12	Instruction page fault
	1	Instruction access fault
	2	Illegal instruction
	0	Instruction address misaligned
	8，9，11	Environment call
	3	Environment break
	3	Load/Store/AMO address breakpoint
	6	Store/AMO address misaligned
	4	Load address misaligned
	15	Store/AMO page fault
	13	Load page fault
	7	Store/AMO access fault
Lowest	5	Load access fault

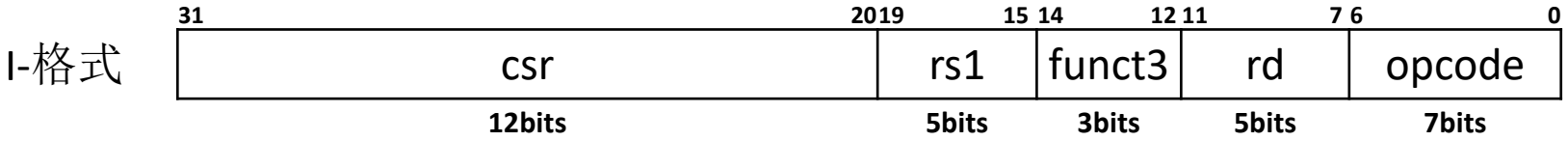
CSR Instruction

- **All CSR instructions atomically read-modify-write a single CSR**
  - whose CSR specifier is encoded in the 12-bit csr field of the instruction held in bits 31–20
    - The immediate forms use a 5-bit zero-extended immediate encoded in the rs1 field.

	31 20	19 15	14 12	11 7	6 0
	csr	rs1	funct3	rd	opcode
	12bits	5bits	3bits	5bits	7bits
I: CSRRW	csr	rs1	001	rd	1110011
I: CSRRS	csr	rs1	010	rd	1110011
I: CSRRC	csr	rs1	011	rd	1110011
I: CSRRWI	csr	uimm	101	rd	1110011
I: CSRRSI	csr	uimm	110	rd	1110011
I: CSRRCI	csr	uimm	111	rd	1110011

CSR传输设置指令

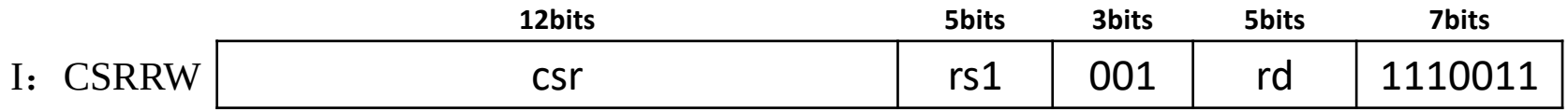
CSR指令格式



12位独立寻址空间支持4096个CSR寄存器

CSRRW -Control and Status Register Read and Write

- Atomic Read and Write CSR
- instruction atomically swaps values in the CSRs and integer registers.
- csrrw rd, csr, rs1
  - ***if*** rd! =x0 then x[rd] ← x[crs]; x[csr] ← x[rs1]



- 伪指令：rd=0，set crs： csrw csr, rs1

CSR传输设置指令

CSRRS -Control and Status Register Read and Set

- Atomic Read and Set Bits in CSR
- instruction reads the value of the CSR, and writes it to integer register rd
- csrrs rd, csr, rs1
  - x[rd] ← x[crs]; ***if*** rs1! =x0 then x[csr] ← {x[rs1] | x[csr]};

I： CSRRS	csr	rs1	010	rd	1110011

- 伪指令 rd =0， set bits in CSR： csrs csr, rs1

CSRRC -Control and Status Register Read and Clear

- Atomic Read and Clear Bits in CSR
- instruction reads the value of the CSR, and writes it to integer register rd
- csrrc rd, csr, rs1
  - x[rd] ← x[crs]; ***if*** rs1! =x0 then x[csr] ← {~x[rs1] **&** x[csr]};

I： CSRRC	csr	rs1	011	rd	1110011

- 伪指令： rs1=0， Clear crs： csrc csr, rs1

CSR立即数传输设置指令

- uimm[4:0] zero-extending a 5-bit unsigned immediate field encoded in the rs1 field instead of a value from an integer register rs1
- **csrrwi rd, csr, uimm[4:0]**
  - Control and Status Register Read and Write Immediate
    - Write CSR伪指令： csrwi csr, uimm[4:0]
- **csrrsi rd, csr, uimm[4:0]**
  - Control and Status Register Set Immediate
    - Set bits CSR伪指令：csrsi csr, uimm[4:0]
- **csrrci rd, csr, uimm[4:0]**
  - Control and Status Register Read and Clear Immediate
    - Clear bit crs伪指令：csrci csr, uimm[4:0]

	31 … 20	19 … 15	14 … 12	11 … 7	6 … 0
I：CSRRWI	csr	uimm	101	rd	1110011
I： CSRRSI	csr	uimm	110	rd	1110011
I： CSRRCI	csr	uimm	111	rd	1110011

Interrupts Instruction

- **Exception return**
  - MRET
    - PC ← MEPC；(MStatus寄存器有变化)

	31–25	24–20	19–15	14–12	11–7	6–0
R：MRET	0011000	00010	00000	000	00000	1110011
R：SRET	0001000	00010	00000	000	00000	1110011
R：wfi	0001000	00101	00000	000	00000	1110011

- **Environment call**
  - ecall
    - MEPC ← PC of ecall instruction itself
- **Breakpoint**
  - ebreak
    - MEPC ← PC of ebreak instruction itself

	31–25	24–20	19–15	14–12	11–7	6–0
I：ecall	0000000	00000	00000	000	00000	1110011
I：ebreak	0000000	00001	00000	000	00000	1110011

RISC-V中断处理--进入异常

- **RISC-V处理器检测到异常，开始进行异常处理：**
  - 停止执行当前的程序流，转而从CSR寄存器mtvec定义的PC地址开始执行；
  - 更新机器模式异常原因寄存器：mcause
  - 更新机器模式中断使能寄存器：mie
  - 更新机器模式异常PC寄存器：mepc
  - 更新机器模式状态寄存器：mstatus
  - 更新机器模式异常值寄存器：mtval

RISC-V中断结构--退出异常

- 异常程序处理完成后，需要从异常服务程序中退出，并返回主程序
  - RISCV中定义了一组退出指令MRET，SRET，和URET
  - 机器模式对应MRET。
- 机器模式下退出异常(MRET)
  - 程序流转而从csr寄存器mepc定义的pc地址开始执行
  - 同时硬件更新csr寄存器机器模式状态寄存器mstatus
    - 寄存器MIE域被更新为当前MPIE的值：mie ← mpie
    - MPIE 域的值则更新为1： MPIE ← 1

Handling Exceptions

- **Save PC of offending (or interrupted) instruction**
  - In RISC-V: Supervisor Exception Program Counter (SEPC)
- **Save indication of the problem**
  - In RISC-V: Supervisor Exception Cause Register (SCAUSE)
  - 64 bits, but most bits unused
    - Exception code field: 2 for undefined opcode, 12 for hardware malfunction, …
- **Jump to handler**
  - Assume at $0000\ 0000\ 1C09\ 0000_{hex}$
    $(\ldots\ldots\ 00011100000010010000000000000000)_2$

    stvec(Supervisor trap handler base address) **?**

An Alternate Mechanism

- **Vectored Interrupts**
  - Handler address determined by the cause
- **Exception vector address to be added to a vector table base register:**
  - Undefined opcode $00\ 0100\ 0000_{two}$
  - Hardware malfunction: $01\ 1000\ 0000_{two}$
  - …: …
- **Instructions either**
  - Deal with the interrupt, or
  - Jump to real handler

?

Handler Actions

- **Read cause, and transfer to relevant handler**
- **Determine action required**
- **If restartable**
  - Take corrective action
  - use SEPC to return to program
- **Otherwise**
  - Terminate program
  - Report error using SEPC, SCAUSE, …

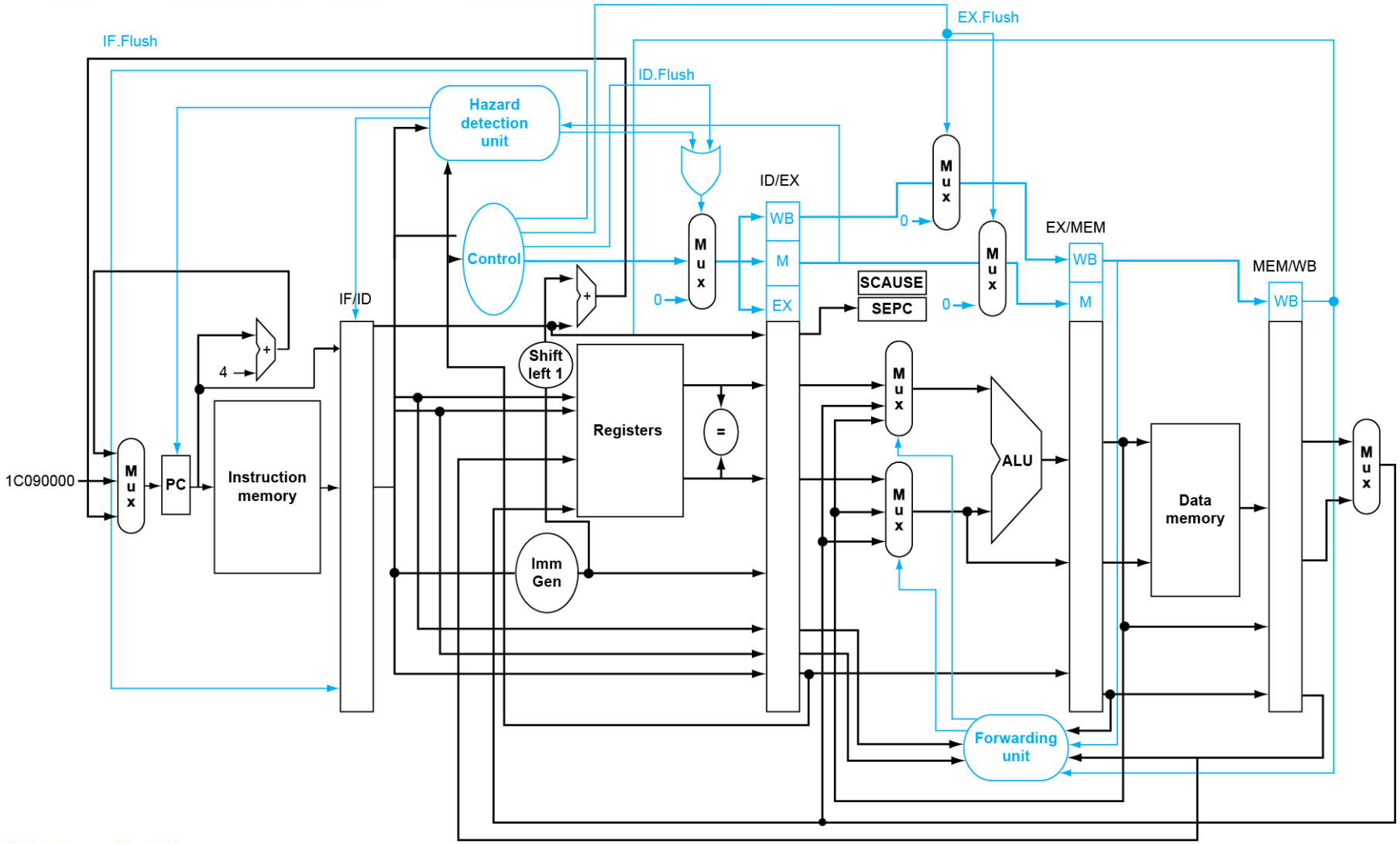
Exceptions in a Pipeline

- **Another form of control hazard**
- **Consider malfunction on add in EX stage**

```
add x1, x2, x1
```

  - Prevent x1 from being clobbered
  - Complete previous instructions
  - Flush `add` and subsequent instructions
  - Set SEPC and SCAUSE register values
  - Transfer control to handler
- **Similar to mispredicted branch**
  - Use much of the same hardware

Pipeline with Exceptions



Exception Properties

- **Restartable exceptions**
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Refetched and executed from scratch
- **PC saved in SEPC register**
  - Identifies causing instruction

Exception Example

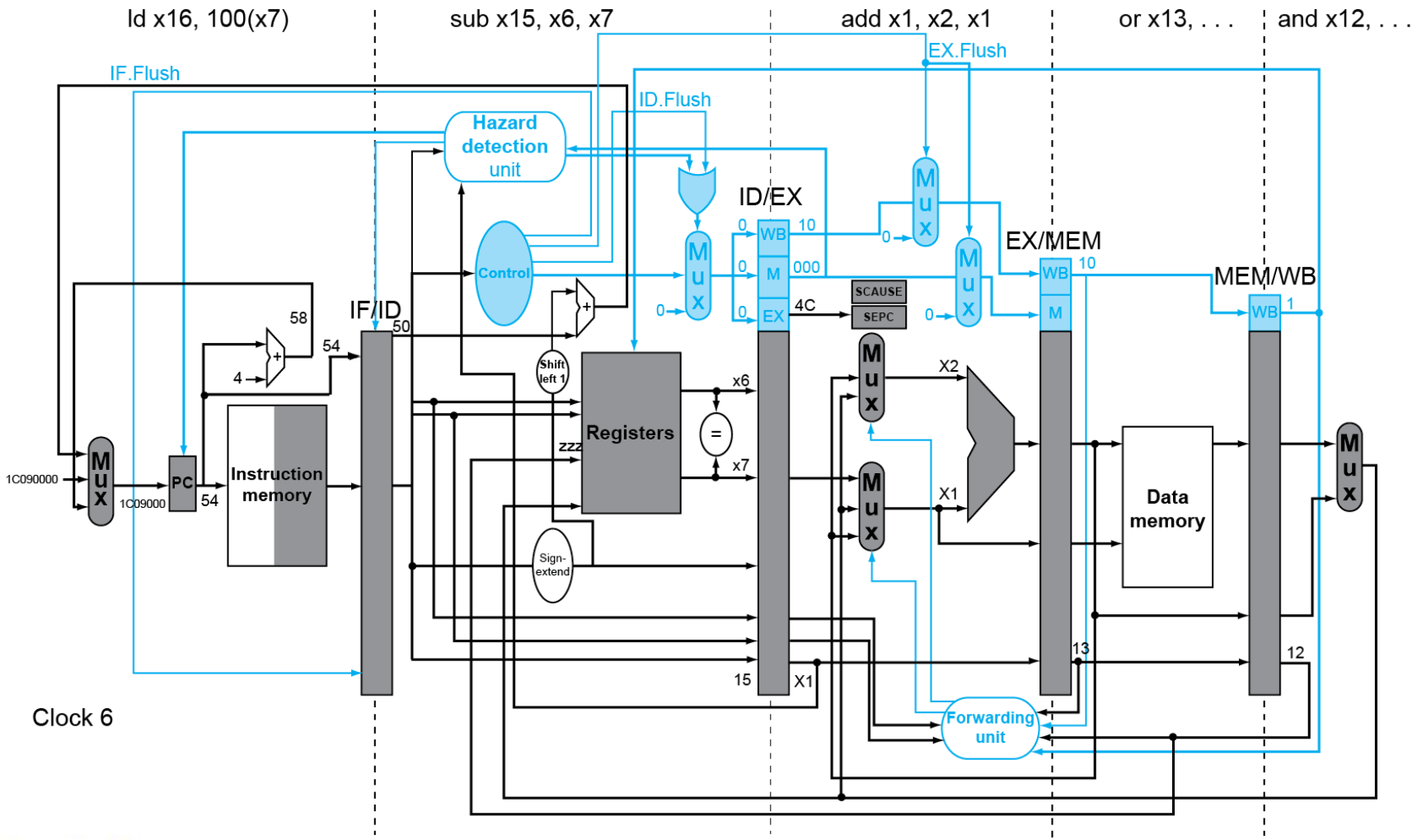
- **Exception on add in**

```
40      sub   x11, x2, x4
44      and   x12, x2, x5
48      orr   x13, x2, x6
4c      add   x1,  x2, x1
50      sub   x15, x6, x7
54      ld    x16, 100(x7)
…
```

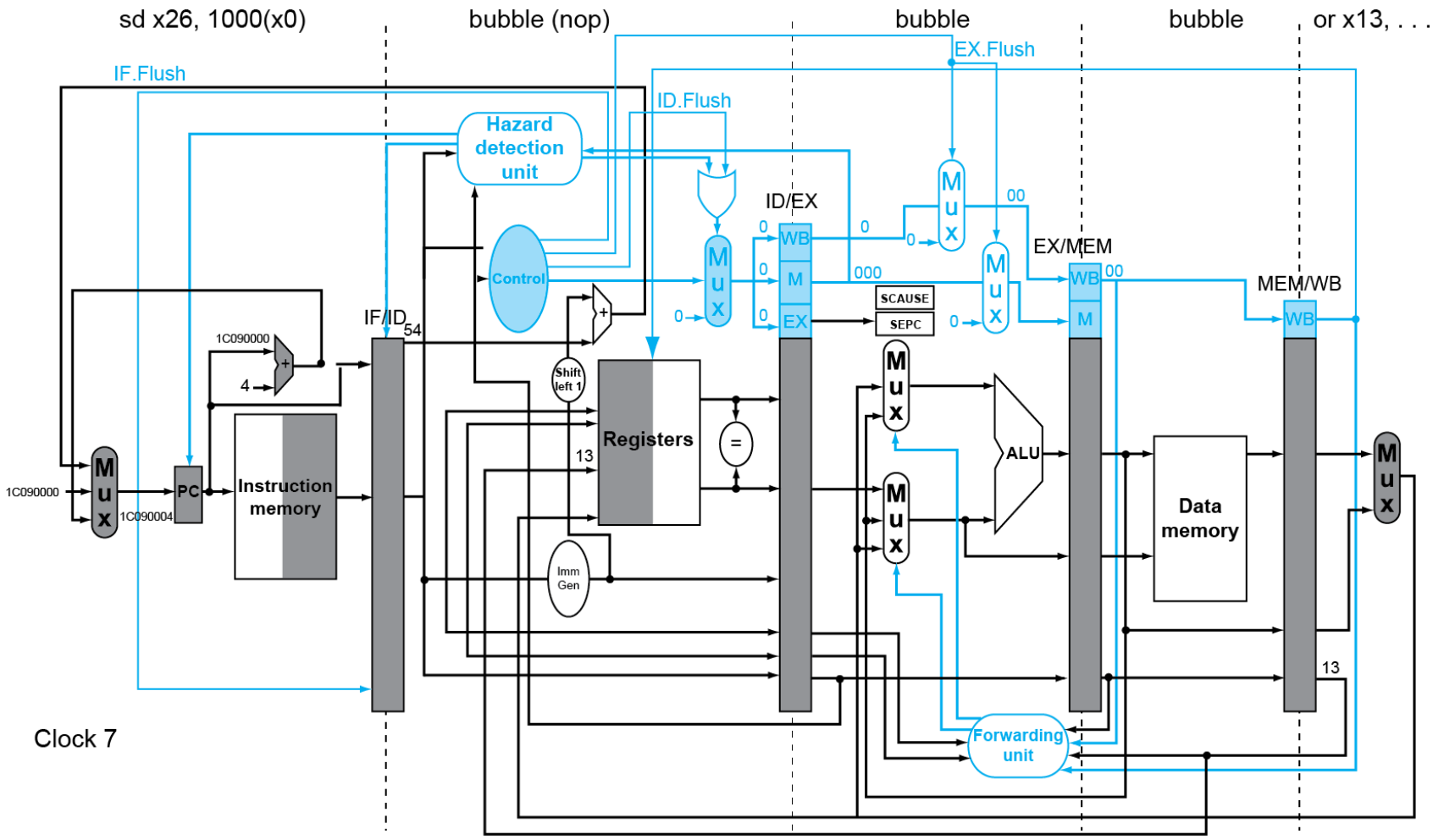
- **Handler**

```
1c090000    sd  x26, 1000(x10)
1c090004    sd  x27, 1008(x10)
…
```

Exception Example



Exception Example



Multiple Exceptions

- **Pipelining overlaps multiple instructions**
  - Could have multiple exceptions at once
- **Simple approach: deal with exception from earliest instruction**
  - Flush subsequent instructions
  - “Precise” exceptions
- **In complex pipelines**
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!

Imprecise Exceptions

- **Just stop pipeline and save state**
  - Including exception cause(s)
- **Let the handler work out**
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require “manual” completion
- **Simplifies hardware, but more complex handler software**
- **Not feasible for complex multiple-issue out-of-order pipelines**

典型处理器中断结构：向量模式

- **Intel x86中断结构**
  - 间接向量：000～3FF，占内存最底1KB空间
    - 每个向量由二个16位生成20位中断地址
    - 共256个中断向量，向量编号n=0～255
    - 分硬中断和软中断，响应过程类同，触发方式不同
    - 硬中断响应由控制芯片8259产生中断号n(接口原理课深入学习)
- **ARM中断结构**
  - 固定向量方式(嵌入式课程深入学习)

异常类型	偏移地址(低)	偏移地址(高)	
复 位	00000000	FFFF0000	
未定义指令	00000004	FFFF0004	
软中断	00000008	FFFF0008	
预取指令终	0000000C	FFFF000C	
数据终止	00000010	FFFF0010	
保留	00000014	FFFF0014	
中断请求(IRQ)	00000018	FFFF0018	
快速中断请求(FIQ)	0000001C	FFFF001C	

Simplify Interrupt Design

简化中断设计

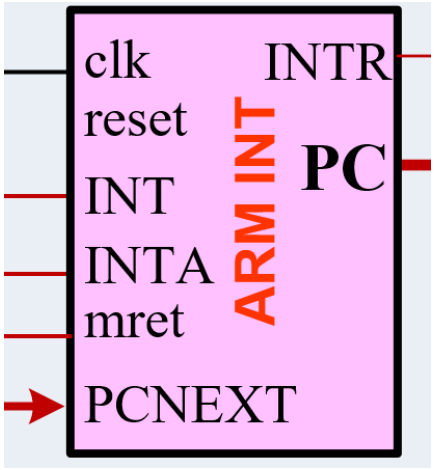
- 采用ARM中断向量(不兼容RISC-V)
  - 实现非法指令异常和外中断
  - 设计EPC
- 兼容RISC-V*
  - 仅M-Mode中断寄存器(MCause、Mstatus、MIE、MIP、MEPC和MTVEC)
  - 设计mret、CSRRW (csrw rd, csr, rs1)和ecall指令

ARM中断向量表

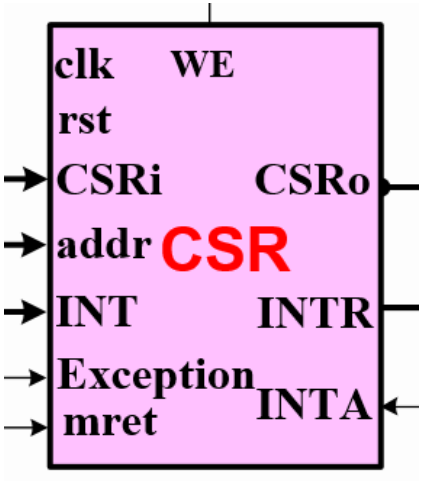
向量地址	ARM异常名称	ARM系统工作模式	本课程定义
0x0000000	复位	超级用户Svc	复位(M-MODE)
0x0000004	未定义指令终止	未定义指令终止Und	非法指令异常
0x0000008	软中断（SWI）	超级用户Svc	ECALL
0x000000c	Prefetch abort	指令预取终止Abt	Int外部中断（硬件）
0x0000010	Data abort	数据访问终止Abt	Reserved自定义
0x0000014	Reserved	Reserved	Reserved自定义
0x0000018	IRQ	外部中断模式IRQ	Reserved自定义
0x000001C	FIQ	快速中断模式FIQ	Reserved自定义

DataPath扩展中断通路

- **DataPath修改**
  - 修改PC模块增加(ARM模式)
    - CPU复位时IE=0，EPC=PC=ox00000000
    - IE=中断使能**(重要)**
    - EPC寄存器，INT触发PC转向中断地址
      - 相当于硬件触发Jal，用eret返回
    - 增加控制信号INT、RFE/eret
      - INT宽度根据扩展的外中断数量设定
  - 修改PC模块增加(RISC-V模式)*
    - CSR简化模块
      - MEPC、MCause和MStatus寄存器等
      - 增加CP0数据通道
        - MEPC、MCause和Status通道
      - 增加控制信号CSR_Write
    - 修改PC通道



ARM中断模块



RV32中断模块

注意：INT是电平信号，不要重复响应

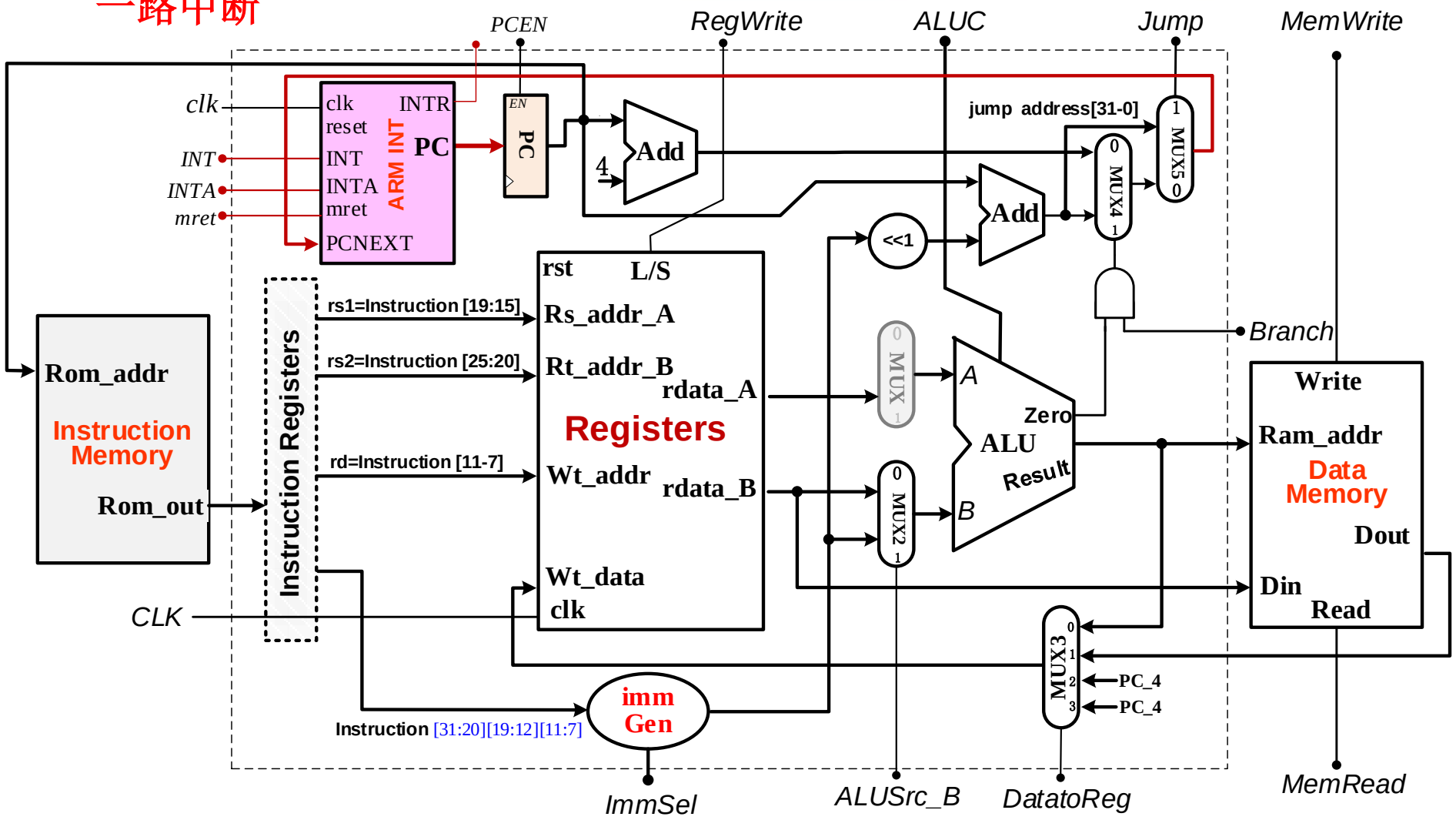
控制器扩展中断译码

- **控制器修改**
  - ARM模式(简单)
    - 仅增加mret指令
    - 中断请求信号触发PC转向，在Datapath模块中修改
  - RISC-V模式(可独立模块)*
    - 至少扩展mret、CSRRW指令译码
    - 增加Wt_Write通道选择控制
    - 增加CSR_Write
    - 增加控制信号mret、ecall、CSR_Write
- **中断调试**
  - 首先时序仿真
  - 物理验证
    - 用BTN[0]触发调试：静态或低速
    - 用计数器counter0_OUT调试：动态或高速

      注意动态测试时死锁!!!

ARM中断模式的DataPath

一路中断



ARM interrupt description

- 中断触发检测与服务锁存

检测与锁存

```
//interrupt Trigger
      assign INTR = int_act;
      assign int_clr = reset| ~int_act;          //clear interrupt Request

      always @(posedge clk)
      INT_get <= {INT_get[0],INT};               //Interrupt Sampling

      always @(posedge clk)begin                 //interrupt Request
         if(INT_get==2'b01)int_req_r<=1;         //set interrupt Request(相当于MEIP)
         else if(int_clr)int_req_r<=0;           //clear interrupt Request
      end
```

- 断点保护、中断开、并与返回

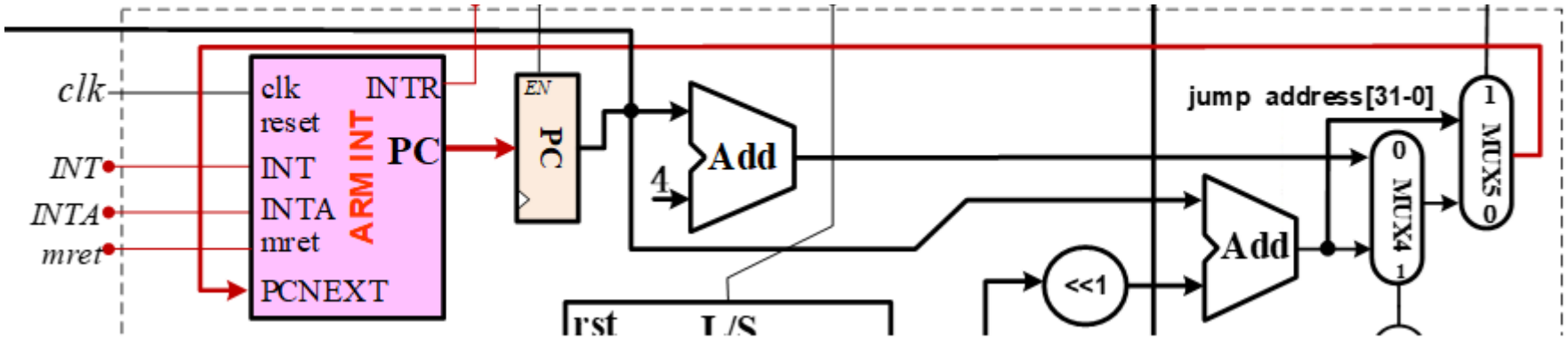
```
always @(posedge clk or posedge reset ) begin
      if (reset)begin EPC     <= 0;            //EPC=32'h00000000;
                      int_act <= 0;
                      int_en  <= 1;            //相当于MIE
      end
      else if(int_req_r & int_en)begin         //int_req_r: interrupt Request reg
                  int_act <= 1;                //interrupt Service set
                  int_en  <= 0;                //interrupt disable
           end
           else begin if(INTA & int_act)
                      begin int_act<=0;
                        EPC <= pc_next;        //interrupt return PC
                        end
                      if(mret) int_en<=1;      //interrupt enable if pc<=EPC;
                  end
end
```

中断通路模块：PC通路

- PC输出通路

仅实现一路中断，请同学们扩展

```
//PC Out
    always @* begin
        if (reset==1)pc <= 32'h00000000;
        else if(INTA)
                pc <= 32'h00000004;        //interrupt Vector    & int_en
            else if(mret)pc <= EPC;        //interrupt return
                else pc <= pc_next;        //next instruction
    end
```

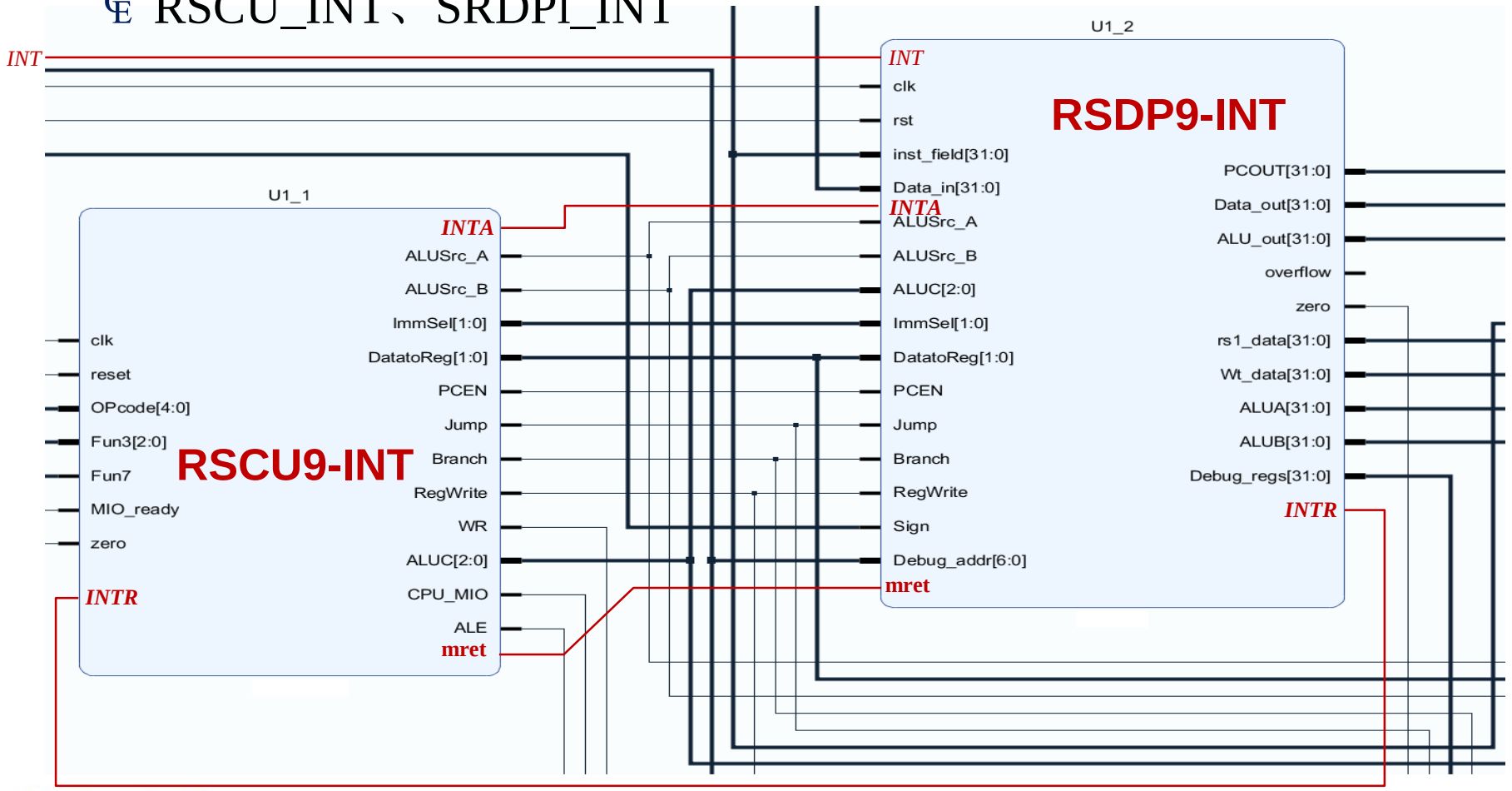


- RSCU_INT ?

INTA ⟷ INTR

增加中断后的CPU模块

- 注意增加信号和模块互连
  - RSCU_INT、SRDPl_INT



修改功能测试程序：判断子代码

```
loop1:
      lw a1, 0x0(s1)          #读GPIO端口F0000000状态，同前。
      add a1, a1, a1;
      add a1, a1, a1;         #左移2位将SW与LED对齐
      sw a1, 0x0(s1);         #再将新$a1:r5写到GPIO端口F0000000,写到LED
      lw a1, 0x0(s1);         #再读GPIO端口F0000000状态
      and s8, a1, t0;         #与80000000相与，即：取最高位=out0,屏蔽其余位
      add s6, s6, t1;         #程序计数延时(加1)
      #beq   s8, t0, C_init;  #硬件计数。out0=1,Counter通道0溢出,转C_init
      #beq s6, zero, C_init;  #若程序计数$t5:r13=0,转C_init
l_next:                       # 延时未到，继续：判断7段码显示模式：SW[4:3]
      lw a1, 0x0(s1) ;        #再读GPIO端口F0000000开关SW
      add s7, a4, a4;         #因x14=4, 故s7：x23=00000008
      add s9, s7, s7;         #s9：x25=00000010
      add s7, s7, s9;         #s7：x23=00000018(00011000)：11对应SWO[4:3]
      and s8, a1, s7;         #取SW[4:3]：屏蔽其余位送x24
      beq s8, zero, L00;      #SW[4:3]=00, L00：7段显示"点"循环移位，SW0=0
      beq s8, s7,    L11;     #SW[4:3]=11，L11：显示七段图形，SW0=0
      add s7, a4, a4 ;        #$s2:r18=8
      beq s8, s7,    L01 ;    #SW[4:3]=01, L01：显示内存预置16进制值
L10：……                      #SW[4:3]=10，L10显示x21(即时值+1)，SW0=1(用户扩展：)
```

SWO状态循环显示

硬件/软件/中断三种定时选择

注释掉

功能判断

修改定时子代码：中断返回

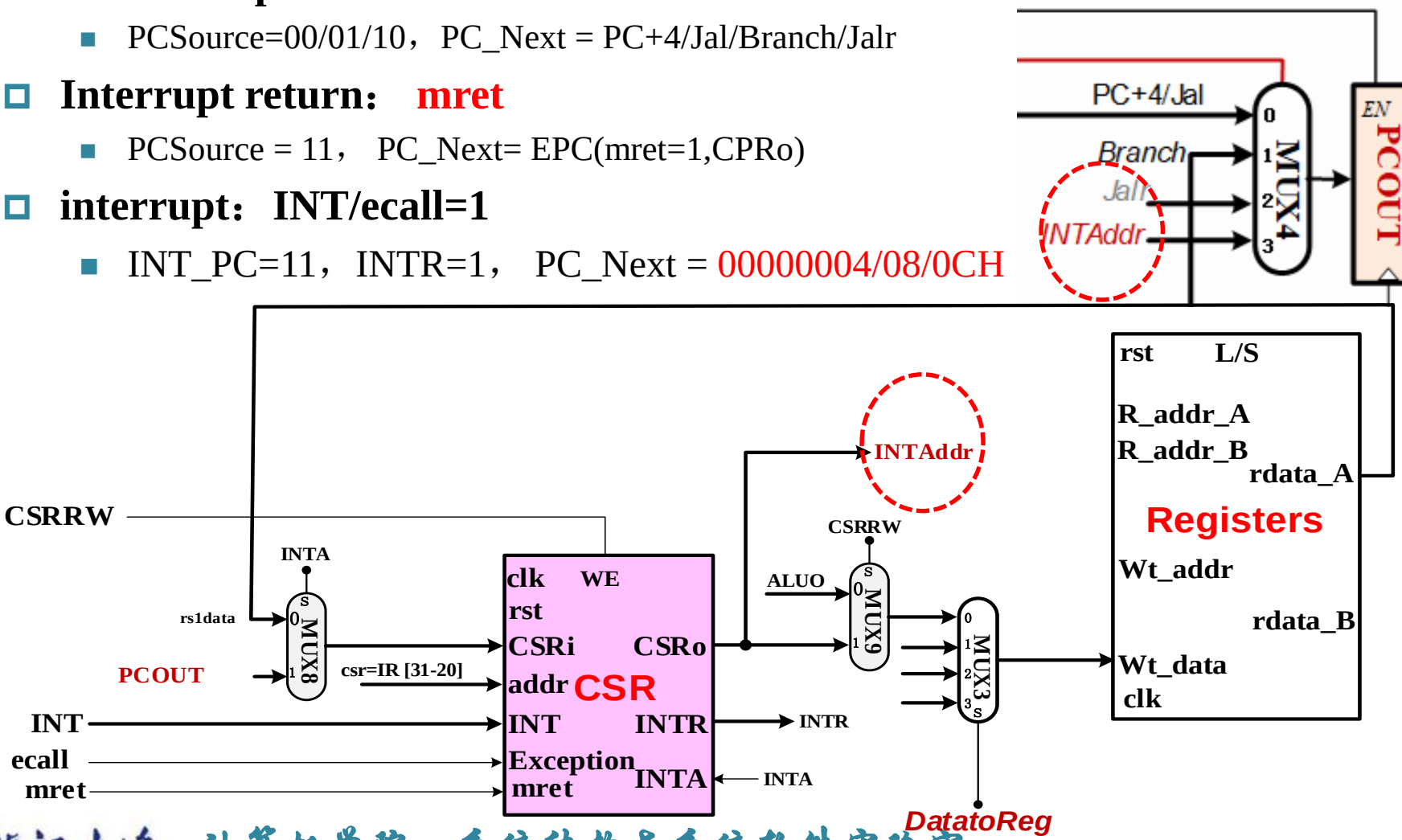
```
C_init:                          # 延时结束，修改显示值和定时/延时初始化
        lw s6, 0x14(zero) ;      # 取程序计数延时初始化常数
        add a5, a5, a5;          # a5左移，x15=xxxxxxx0，七段图形点左移
        or a5, a5, t1;           # a5:x15末位置1，消除七段显示器右上角点，不显示
        add s3, s3, a4;          # x14=00000004，LED图形访存地址+4
        and s3, s3, s0;          # 和3F相与，x19=000000xx，屏蔽高位，简单截取低位地址(6位)
        add s5, s5, t1 ;         # x21+1
        beq s5, a3, L6 ;         #若x21=ffffffff，重置x21=5
        j L7;
L6:
        add s5, zero, a4 ;       #x21=4
        add s5, s5, t1 ;         #重置x21=5
L7:
        lw a1, 0x0(s1) ;         #读GPIO端口F0000000状态
        add s8, a1, a1;
        add s8, s8, s8 ;         # 左移2位将SW与LED对齐，同时D1D0置00，选择计数器通道0
        sw s8, 0x0(s1);          # x24输出到GPIO端口F0000000，计数器通道counter_set=00端口
                                 不变、LED=SW：  {GPIOf0[15:2], LED, GPIOf0[1:0]/counter_set}
        sw a2, 0x4(s1)           # 计数器端口:F0000004，送计数常数x12=F8000000
        mret;                    #本处直接跳转，若中断或子程序则调用则返回
```

注：硬件计数与计数中断时要判断硬件计数溢出已经消除，否则会造成多次进入。

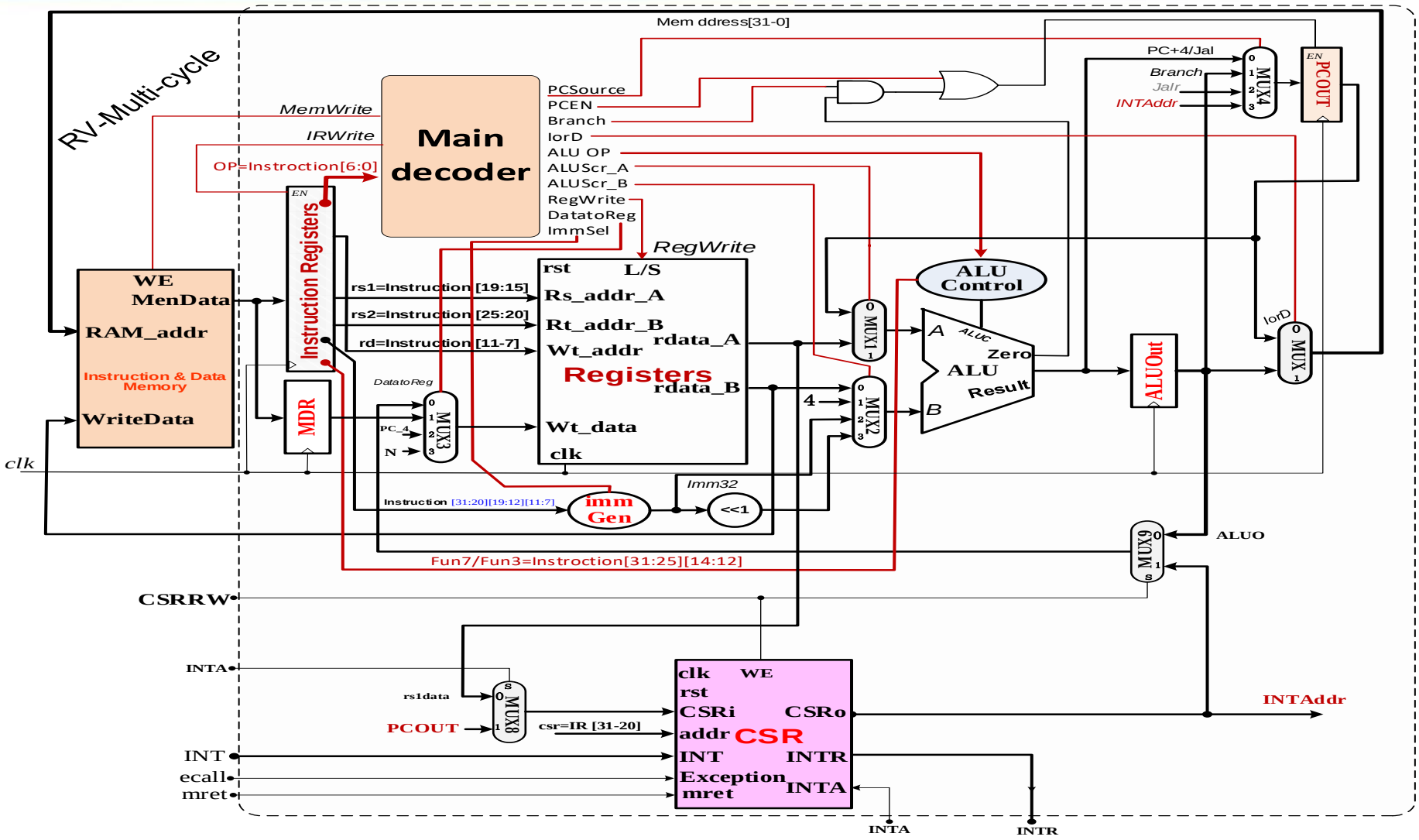
注意动态测试时死锁!!!

CSR Datapath with Multi-cycle

- **No interruption：Normal execution**
  - PCSource=00/01/10，PC_Next = PC+4/Jal/Branch/Jalr
- **Interrupt return： mret**
  - PCSource = 11， PC_Next= EPC(mret=1,CPRo)
- **interrupt：INT/ecall=1**
  - INT_PC=11，INTR=1， PC_Next = 00000004/08/0CH



Interruption Implementation



CP0 Interrupt Description

```
1  module      CP0(input  clk,rst,
2                  input  INTA,
3                  input  [4:0] addr,
4                  input  [31:0]CPRi,
5                  input  WE,
6                  input  [7:0]INT,
7                  input  Exception,
8                  input  eret,
9                  output [31:0] CPRo,
10                 output [31:0] Cause,
11                 output [31:0] Status,
12                 output INTR );
13 reg [31:0] CP0 [0:3];                          // CP0.0-CP0.3
14 reg [7:0]Trig;
15 reg [1:0] req_get;
16 reg int_clr,int_act;
17 integer i;
18
19    wire [4:0] Exc_code = {1'b0,Exception,3'b000};
20    assign EPC  = CP0[2];
21    assign Cause   = CP0[1];
22    assign Status  = CP0[0];
23    assign int_en  = Status[0];
24    assign INTR = int_act;
25    wire [7:0] INT_MK   = INT & Status[15:8];          //TInterrupt mask
26    assign int_req = | {INT_MK, Exception };
27    always @(posedge clk)
28       req_get <= {req_get[0], int_req};              //外部中断采样
```

从这里我们可以得出什么启发？

这是MIPS中断CP0简化描述。CSR相当于CP0的变形，请同学根据CRS相关中断寄存器定义修改CP0实现RISC-V m-mode简化中断。

ExcCode：sys=08=5’b01000

CP0 Interrupt Description-1

```
29  //interrupt Trigger
30     always @(posedge clk)
31        if(rst)
32           int_act <= 0;
33        else if(int_en && (req_get=2'b01))
34                 int_act <= 1;                          //*interrupt Service
35              else if(INTA & int_act)                   // ACK interrupt Service
36                  int_act<=0;                           // clear interrupt Request
37
38  // CP0 Register Access
39     assign CPRo =(~eret) ? CP0[{addr[1:0]}] : EPC ;    // read
40
41     always @(posedge clk or posedge rst) begin
42           if (rst==1) begin                            // reset
43           int_clr <=0 ;
44           CP0[0] <= 32'h0000FF00;
45           for (i=1; i<4; i=i+1)CP0[i] <= 0;            //CP0 clear;
46        end
47        else begin
48             if ((addr[4:2] == 3'b011) &&(WE == 1))     // write
49                 CP0[{addr[1:0]}] <= CPRi;
50              else begin
51                 if(INTA | Exception)begin
52                 CP0[2]  <= CPRi;                  //Save interrupt return PC
53                 CP0[1]  <= {Cause[31:16],INT_MK,1'b0,Exc_code,2'b00};
54                 CP0[0]  <= {Status[31:1], 1'b0};
55                 end
56                 if(eret) CP0[0] <= {Status[31:1], 1'b1};    //restore interrupt enable if pc<=EPC
57              end
58        end
59     end
60  endmodule
```

Discussion in Class

- **Multiple Exceptions in RISC-V Pipeline**

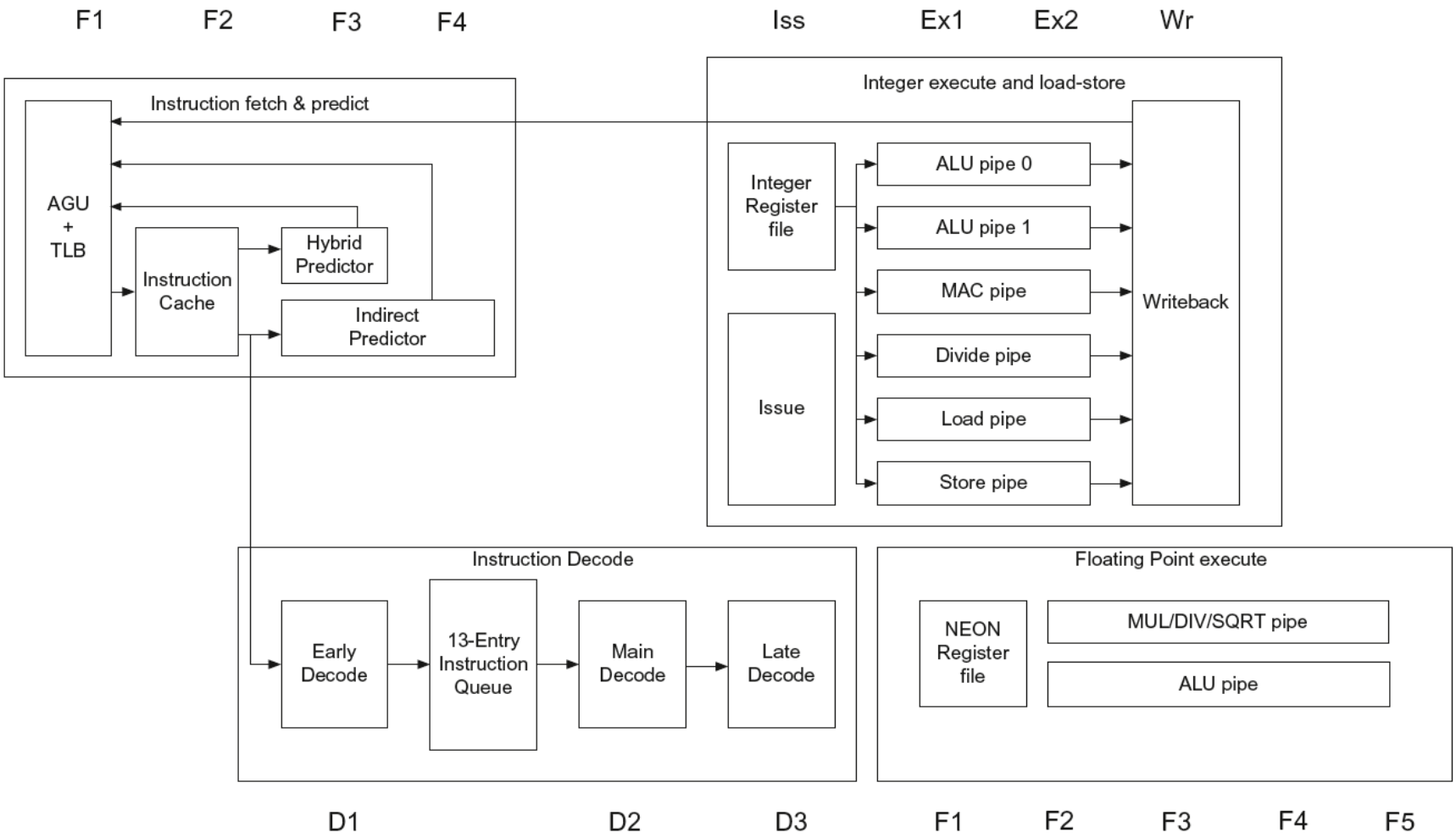
How to Implementation ?

What is commit ?

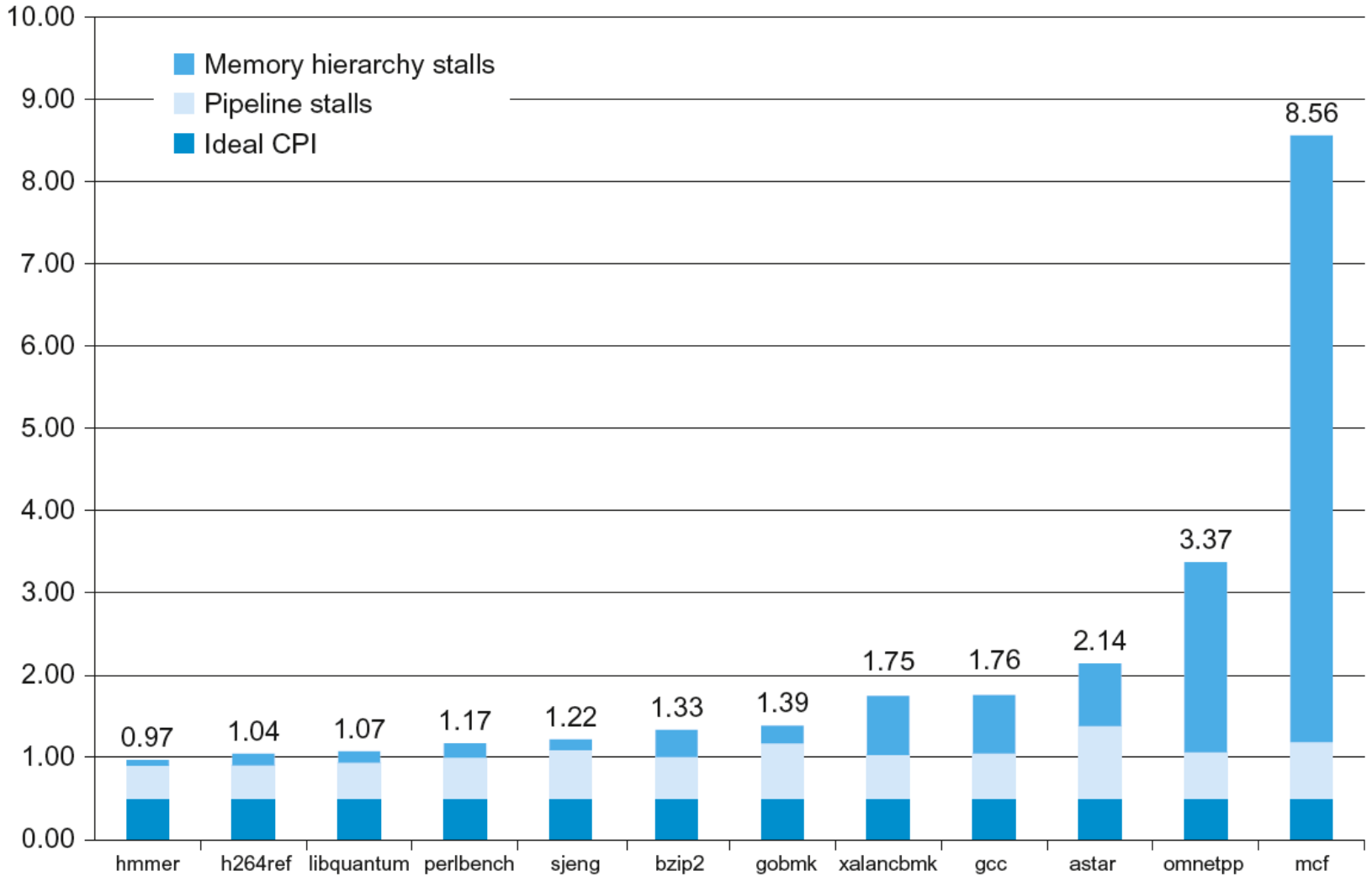
Cortex A53 and Intel i7

Processor	ARM A53	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	100 milliWatts (1 core @ 1 GHz)	130 Watts
Clock rate	1.5 GHz	2.66 GHz
Cores/Chip	4 (configurable)	4
Floating point?	Yes	Yes
Multiple issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline stages	8	14
Pipeline schedule	Static in-order	Dynamic out-of-order with speculation
Branch prediction	Hybrid	2-level
1st level caches/core	16-64 KiB I, 16-64 KiB D	32 KiB I, 32 KiB D
2nd level caches/core	128-2048 KiB	256 KiB (per core)
3rd level caches (shared)	(platform dependent)	2-8 MB

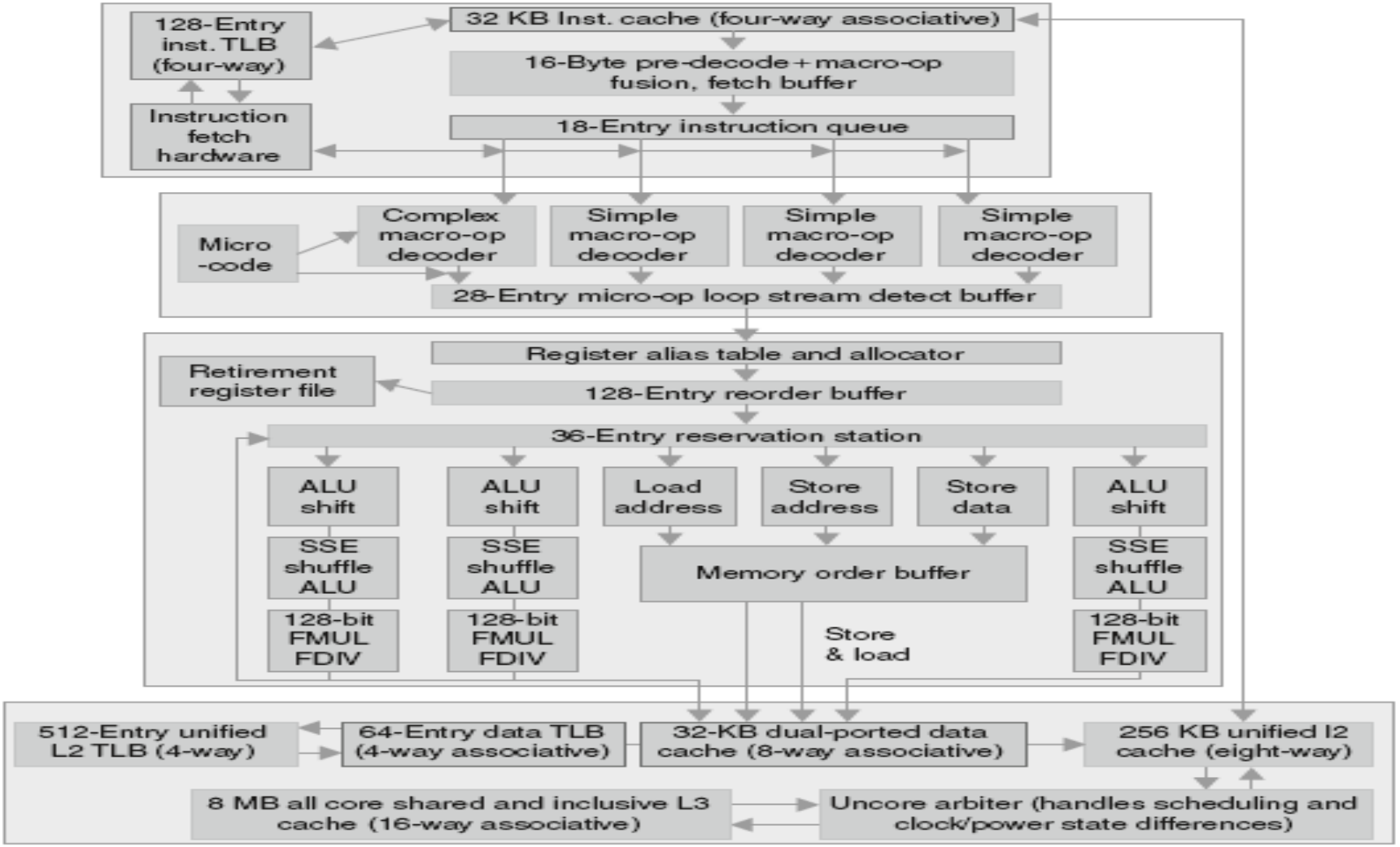
ARM Cortex-A53 Pipeline



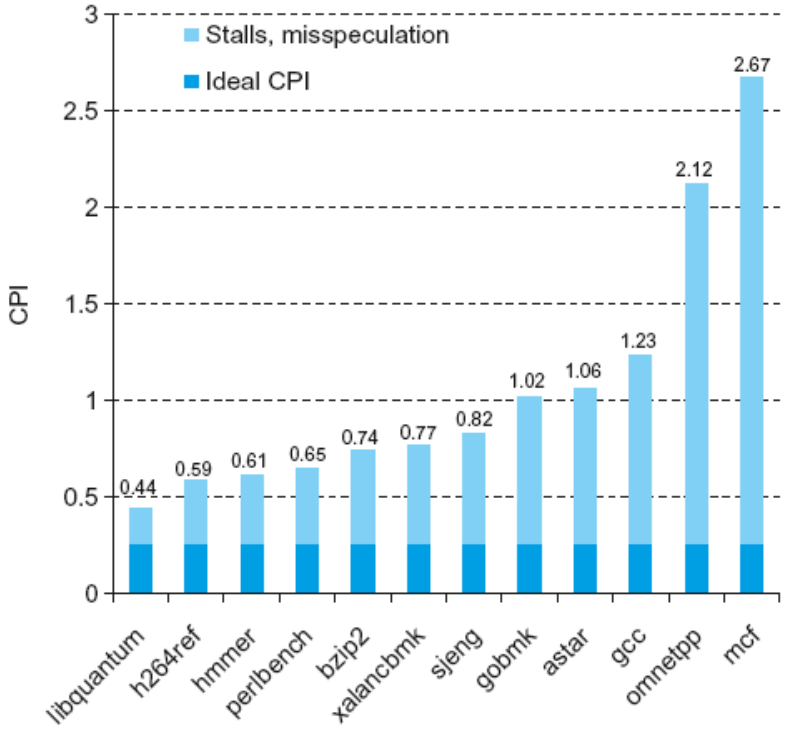
ARM Cortex-A53 Performance

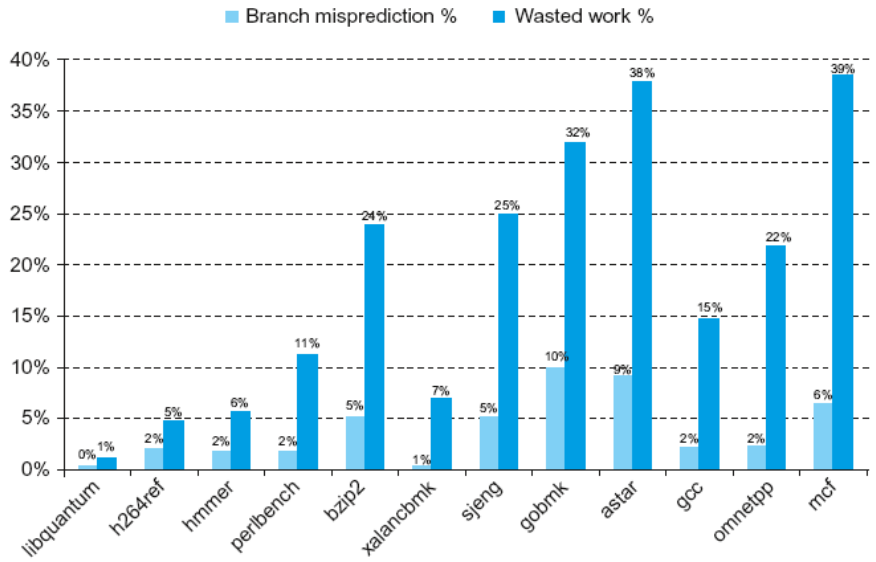


Core i7 Pipeline



Core i7 Performance





Fallacies

- **Pipelining is easy (!)**
  - The basic idea is easy
  - The devil is in the details
    - e.g., detecting data hazards
- **Pipelining is independent of technology**
  - So why haven't we always done pipelining?
  - More transistors make more advanced techniques feasible
  - Pipeline-related ISA design needs to take account of technology trends
    - e.g., predicated instructions

Pitfalls

- **Poor ISA design can make pipelining harder**
  - e.g., complex instruction sets (VAX, IA-32)
    - Significant overhead to make pipelining work
    - IA-32 micro-op approach
  - e.g., complex addressing modes
    - Register update side effects, memory indirection
  - e.g., delayed branches
    - Advanced pipelines have long delay slots

Concluding Remarks

- **ISA influences design of datapath and control**
- **Datapath and control influence design of ISA**
- **Pipelining improves instruction throughput using parallelism**
  - More instructions completed per second
  - Latency for each instruction not reduced
- **Hazards: structural, data, control**
- **Multiple issue and dynamic scheduling (ILP)**
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall


ZHEJIANG UNIVERSITY
1897


END

附录

- MIPS interrupt

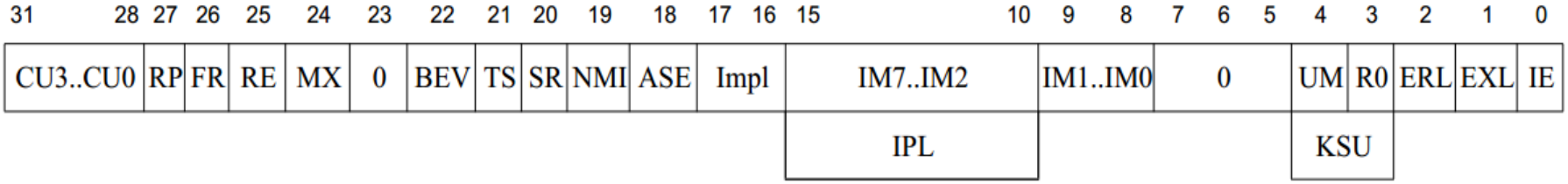
MIPS interrupt structure

- **Coprocessor 0 : CP0**
  - 32 Registers number(5bit indication)
  - Each number contains 8 registers(: Seletion 3bit)
    - Each register consists of 32 bits
- **CP0 register**

Register	number	function
BadVAddr	8	最近内存访问异常的地址
Count	9	高精度内部计时计数器
Compare	11	定时常数匹配比较寄存器
Status(SR)	**12**	**状态寄存器、特权、中断屏蔽及使能等，可位控**
Cause	**13**	**中断异常类型及中断持起位**
EPC	**14**	**中断返回地址**
Config	16	配置寄存器，依赖于具体系统
……		

Status Register (SR)

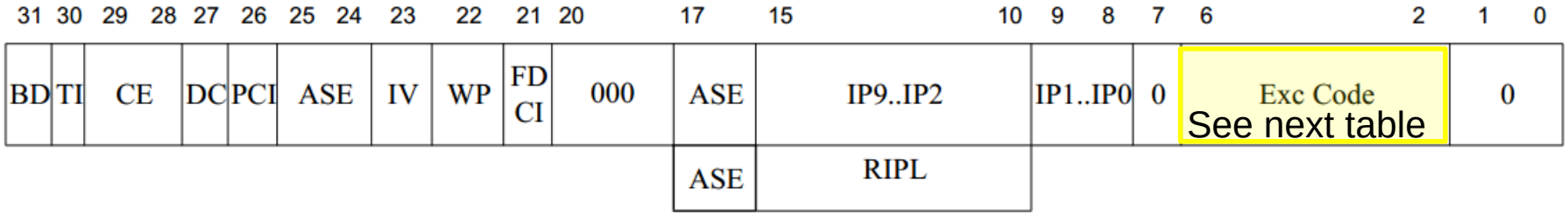
CP0[12]



Name	Bits	Description	Read/Write
CU3-0	31-28	协处理器存在标志 （0:不存在 1:存在）	R/W
BEV	**22**	**中断向量行为 （0:Normal 1:Bootstrap）**	**R/W**
SR	20	上电复位方式 （0:硬复位 1:软复位）	R/W
IM7-0	**15-8**	**中断允许位 （0:屏蔽 1:允许）**	**R/W**
IPL	15-10	当前中断优先级 （仅EIC模式）	R/W
UM	4	当前运行级别 （0:内核态 1:用户态）	R/W
ERL	2	错误标志 （0:正常 1:错误），在上电/复位时置位	R/W
EXL	1	异常标志 （0:正常 1:异常），在内部/外部/软件中断时置位	R/W
IE	**0**	**全局中断允许位 （0:屏蔽 1:允许）**	**R/W**

Cause Register

CP0[13]



Name	Bits	Description	R/W
BD	31	中断是否发生在跳转延时槽中	R
TI	30	时钟中断标志	R
PCI	26	性能计数器中断标志	R
IV	**23**	**是否使用特殊向量地址，详见手册**	**R/W**
IP7-0	15-8	当前中断等待标志，对应兼容和向量模式的8个中断号	R
RIPL	15-10	当前正在执行的中断号，仅外部中断控制模式	R
Exc Code	**6-2**	**CPU内部异常号，** See next table	**R**

Interrupt/Exception coding

ExcCode	名称	异常产生原因
0	**Int**	**外中断（硬件）**
4	AdEL	地址错误异常（L/S)
5	AdES	地址错误异常（存储)
6	IBE	取指令的总线错误
7	DBE	L/S的总线错误
8	**Sys**	**系统调用异常**
9	Bp	断点异常
10	**RI**	**非法指令异常**
11	CpU	没有实现的协处理器
12	**Ov**	**算术上溢异常**
13	Tr	陷阱
15	FPE	浮点
……		

CP0 transfer instructions

Control register access instructions

- Read CP0 instruction：mfco
  - mfc0 rt,rd：GPR[rt] ← CP0[rd]

Op=6bit	rs=00000	rt=5bit	Rd=5bit	=11个0
0x10	0	rt	rd	00000 000000

- Write CP0 instruction：mtco指令
  - mtc0 rd，rt：GPR[rd] ← CP0[rt]

Op=6bit	rs=00000	rt=5bit	Rd=5bit	=11位0
0x10	4	rt	rd	00000 000000

Interrupt related instructions

Exception return

- eret
  - PC ← EPC；(CP0的Cause和Status寄存器有变化)

Op=6bit	1	19bit	FUN
0x0	1	000 0000 0000 0000 0000	011000

System call

- syscall
  - EPC = PC+4; PC ← 异常处理地址; Cause和Status寄存器有变化
- 参数：
  - $v0=系统调用号：Fig B-9-1
  - $a0~$a3、$f12，返回在$v0

Op=6bit	20bit	FUN
0x0	0000 0000 0000 0000 0000	0011000

MIPS Interrupt response

- **Interruption initialization** CP0[12]
  - Set SR：Disable interrupts IE=0，KSU ERL EXL=00 0 0
  - system initialization
  - Set SR： Enable interrupt IE=1，设置IM7-0
- **Interrupt response**
  - **Save Breakpoint by HW**： EPC ← PC+4 CP0[14]
  - **Modify PC by HW**： **PC ← Vector address，Disable interrupts**
  - **Interrupt service**： **Saved register**、 ***Enable interrupt****、 **Service routine** 、**Restore register**
  - Open interrupt：Enable interrupt*
  - Return：eret （**Modify** Cause & Status register by HW）

How Exceptions Are Handled

Design

- add a register: exception program counter(EPC)

  save the address of the offending instruction

- add a status register: cause register(CauseReg)

  hold a field that indicates the reason for the exception.

$$\text{bit0} = \begin{cases} 0 & \text{undefined instruction} \quad \text{ExcCode=10} \\ 1 & \text{overflow} \quad \text{ExcCode=12} \end{cases}$$

- *Another method is to use vector interrupts*

Exception type	vector address
undefine instr	$c0\ 00\ 00\ 00_H$
overflow	$c0\ 00\ 00\ 20_H$

Exp08：00000004H

MIPS MMU

See MIPS Run Linux

System restart

B=*101***1** 0x1FFFFFFF

A=*101***0** 0x00000000

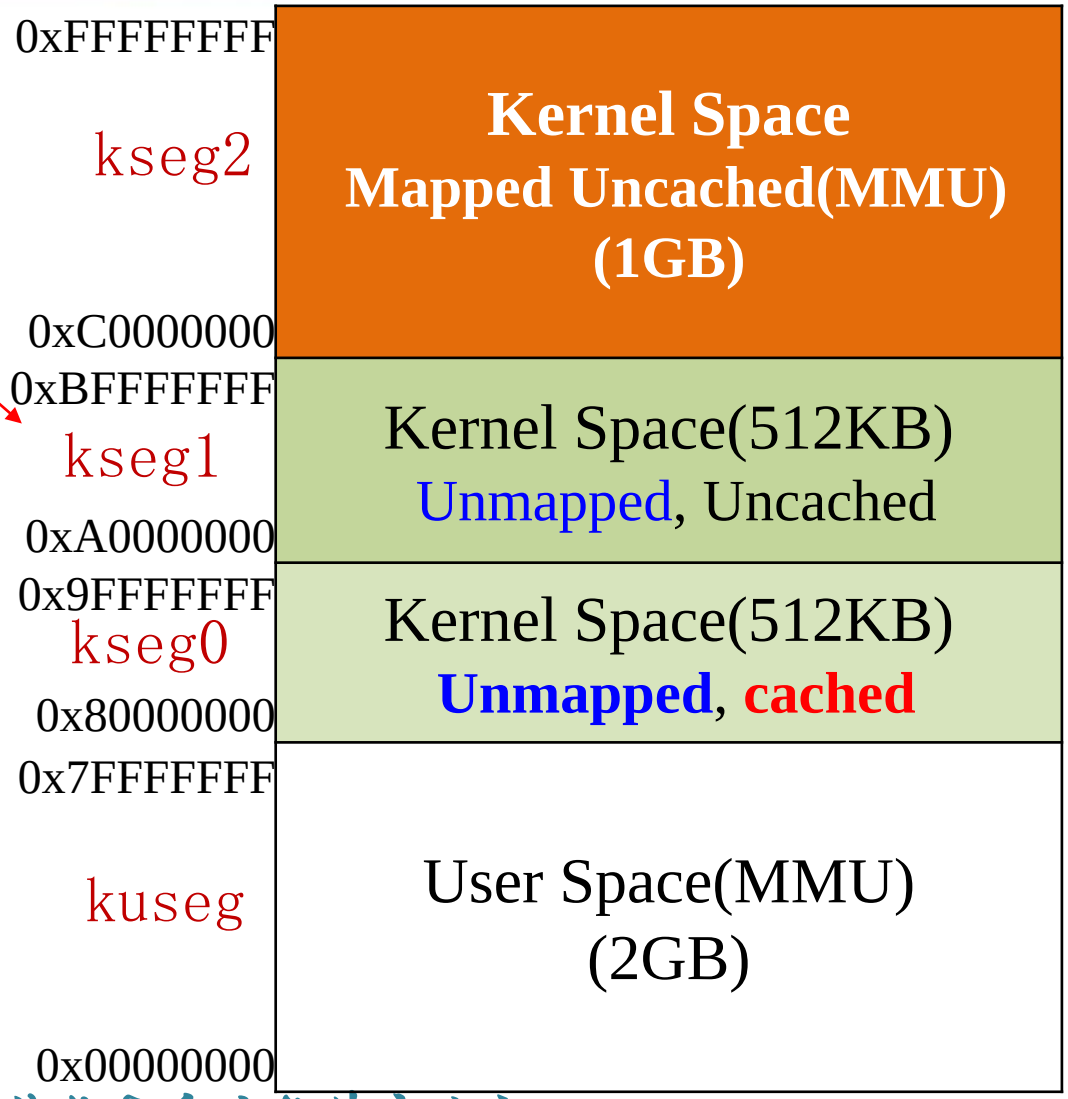
9=*1***001** 0x1FFFFFFF

0=*1***000** 0x00000000

System restart：

PC=0x**B**FC0 0000

高3位清零 =0x**1**FC0 0000



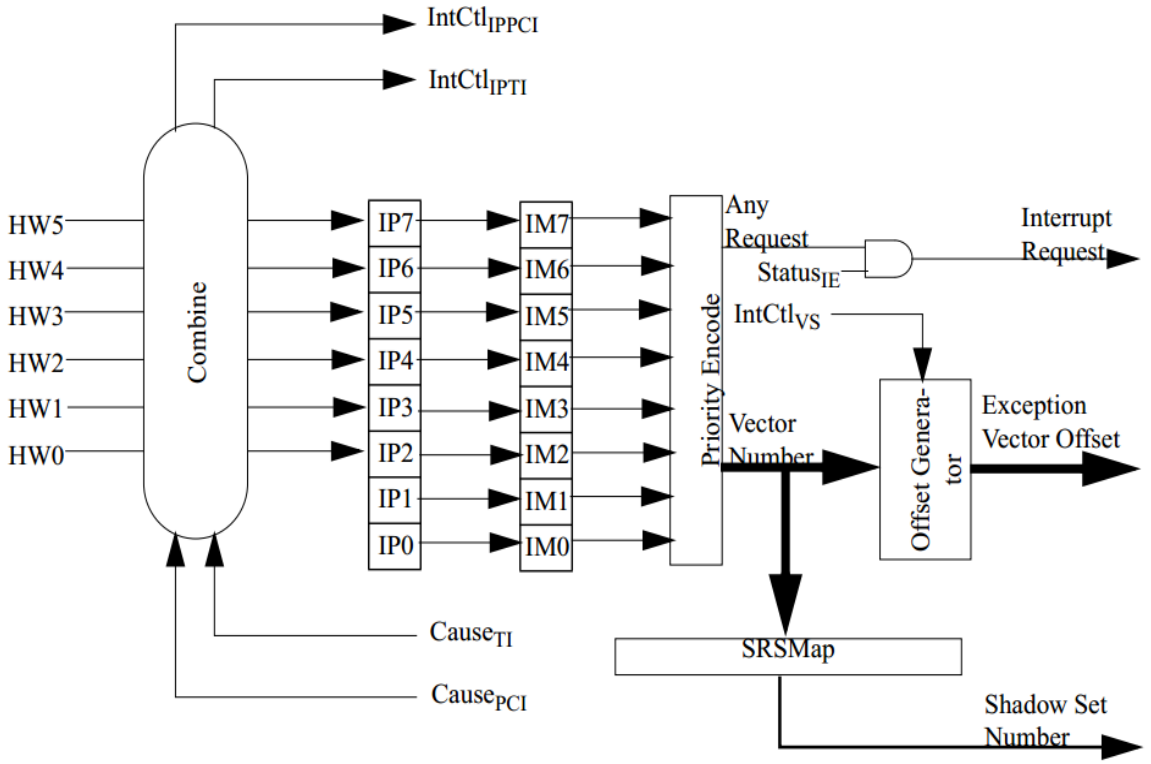
MIPS interrupt mode： Compatibility

- 兼容模式（Interrupt Compatibility Mode）
  - 6个硬件中断号和2个软件中断号
  - 非向量：使用统一的入口地址
    - 软件查询中断源

中断类型	中断源	中断请求源
硬件中断、时钟中断或性能计数中断	HW5	$Cause_{IP7}$ & $Stavus_{IM7}$
硬件中断	HW4	$Cause_{IP6}$ & $Stavus_{IM6}$
	HW3	$Cause_{IP5}$ & $Stavus_{IM5}$
	HW2	$Cause_{IP4}$ & $Stavus_{IM4}$
	HW1	$Cause_{IP3}$ & $Stavus_{IM3}$
	HW0	$Cause_{IP2}$ & $Stavus_{IM2}$
软件中断	SW1	$Cause_{IP1}$ & $Stavus_{IM1}$
	SW0	$Cause_{IP0}$ & $Stavus_{IM0}$

MIPS中断：向量模式一

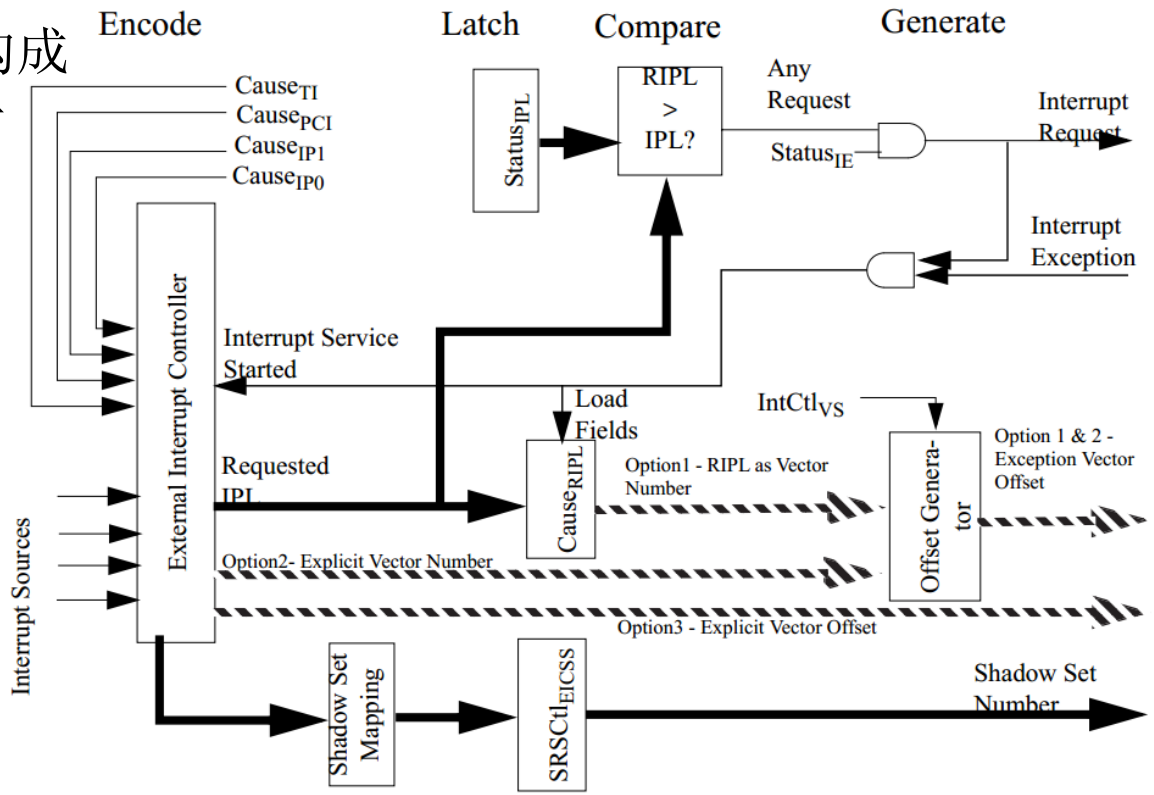
- **中断向量模式(Vectored Interrupt Mode)**
  - 6个硬件中断号和2个软件中断号
  - 由其中一个决定起始向量地址：有多个时由硬件决定那一个
  - 按向量递增方式决定入口地址



MIPS中断：向量模式二

- 外部控制模式(External Interrupt Controller Mode)
  - 由外部控制器提供入口地址，甚至可编程控制
  - 6位中断号
    - 由个硬件中断号构成
    - 1-63共63个中断号
    - 0表示没有中断

Vector Number	Value of $IntCtl_{VS}$ Field				
	0b00001	0b00010	0b00100	0b01000	0b10000
0	0x0200	0x0200	0x0200	0x0200	0x0200
1	0x0220	0x0240	0x0280	0x0300	0x0400
2	0x0240	0x0280	0x0300	0x0400	0x0600
3	0x0260	0x02C0	0x0380	0x0500	0x0800
4	0x0280	0x0300	0x0400	0x0600	0x0A00
5	0x02A0	0x0340	0x0480	0x0700	0x0C00
6	0x02C0	0x0380	0x0500	0x0800	0x0E00
7	0x02E0	0x03C0	0x0580	0x0900	0x1000
•					
61	0x09A0	0x1140	0x2080	0x3F00	0x7C00
62	0x09C0	0x1180	0x2100	0x4000	0x7E00
63	0x09E0	0x11C0	0x2180	0x4100	0x8000




```
vectorOffset ← 0x200 + (vectorNumber × (IntCtl_VS ‖ 0b00000))
```

MIPS中断模式选择

BEV Stavus	IV Cause	VS IntCtl	VINT $Config_3$	VEIC $Config_3$	中断模式	说明
1	x	x	x	x	兼容模式	启动向量
x	0	x	x	x	兼容模式	“非向量”
x	x	=0	x	x	兼容模式	“非向量”
0	1	≠0	1	0	向量模式	专用向量
0	1	≠0	x	1	外部向量	专用向量
0	1	≠0	0	0	不允许	

典型处理器中断结构：向量模式

- **Intel x86中断结构**
  - 间接向量：000～3FF，占内存最底1KB空间
    - 每个向量由二个16位生成20位中断地址
    - 共256个中断向量，向量编号n=0～255
    - 分硬中断和软中断，响应过程类同，触发方式不同
    - 硬中断响应由控制芯片8259产生中断号n(接口原理课深入学习)
- **ARM中断结构**
  - 固定向量方式(嵌入式课程深入学习)

异常类型	偏移地址(低)	偏移地址(高)	
复 位	00000000	FFFF0000	
未定义指令	00000004	FFFF0004	
软中断	00000008	FFFF0008	
预取指令终	0000000C	FFFF000C	
数据终止	00000010	FFFF0010	
保留	00000014	FFFF0014	
中断请求(IRQ)	00000018	FFFF0018	
快速中断请求(FIQ)	0000001C	FFFF001C	

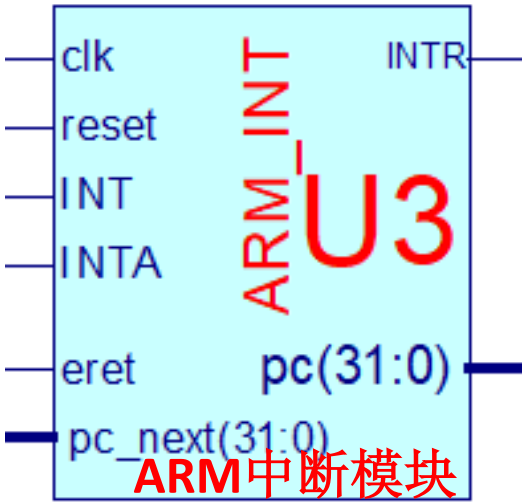
Simplify Interrupt Design

- 简化中断设计
  - 采用ARM中断向量(不兼容MIPS)
    - 实现非法指令异常和外中断
    - 设计EPC
  - 兼容MIPS*
    - Cause和Status寄存器
    - 设计mfc0、mtc0指令
- ARM中断向量表

向量地址	ARM异常名称	ARM系统工作模式	本实验定义
0x0000000	复位	超级用户Svc	内核模式
0x0000004	未定义指令终止	未定义指令终止Und	RI内核模式
0x0000008	软中断（SWI）	超级用户Svc	Sys系统调用
0x000000c	Prefetch abort	指令预取终止Abt	Reserved自定义
0x0000010	Data abort	数据访问终止Abt	Ov
0x0000014	Reserved	Reserved	Reserved自定义
0x0000018	IRQ	外部中断模式IRQ	Int外中断（硬件）
0x000001C	FIQ	快速中断模式FIQ	Reserved自定义

DataPath扩展中断通路

- **DataPath修改**
  - 修改PC模块增加(ARM模式)
    - CPU复位时IE=0，EPC=PC=ox00000000
    - IE=中断使能(重要)
    - EPC寄存器，INT触发PC转向中断地址
      - 相当于硬件触发Jal，用eret返回
    - 增加控制信号INT、RFE/eret
      - INT宽度根据扩展的外中断数量设定
  - 修改PC模块增加(MIPS模式)*
    - CP0简化模块
      - EPC、Cause和Status寄存器
      - 增加CP0数据通道
        - EPC、Cause和Status通道
      - 增加控制信号CP0_Write
    - 修改PC通道



注意：INT是电平信号，不要重复响应

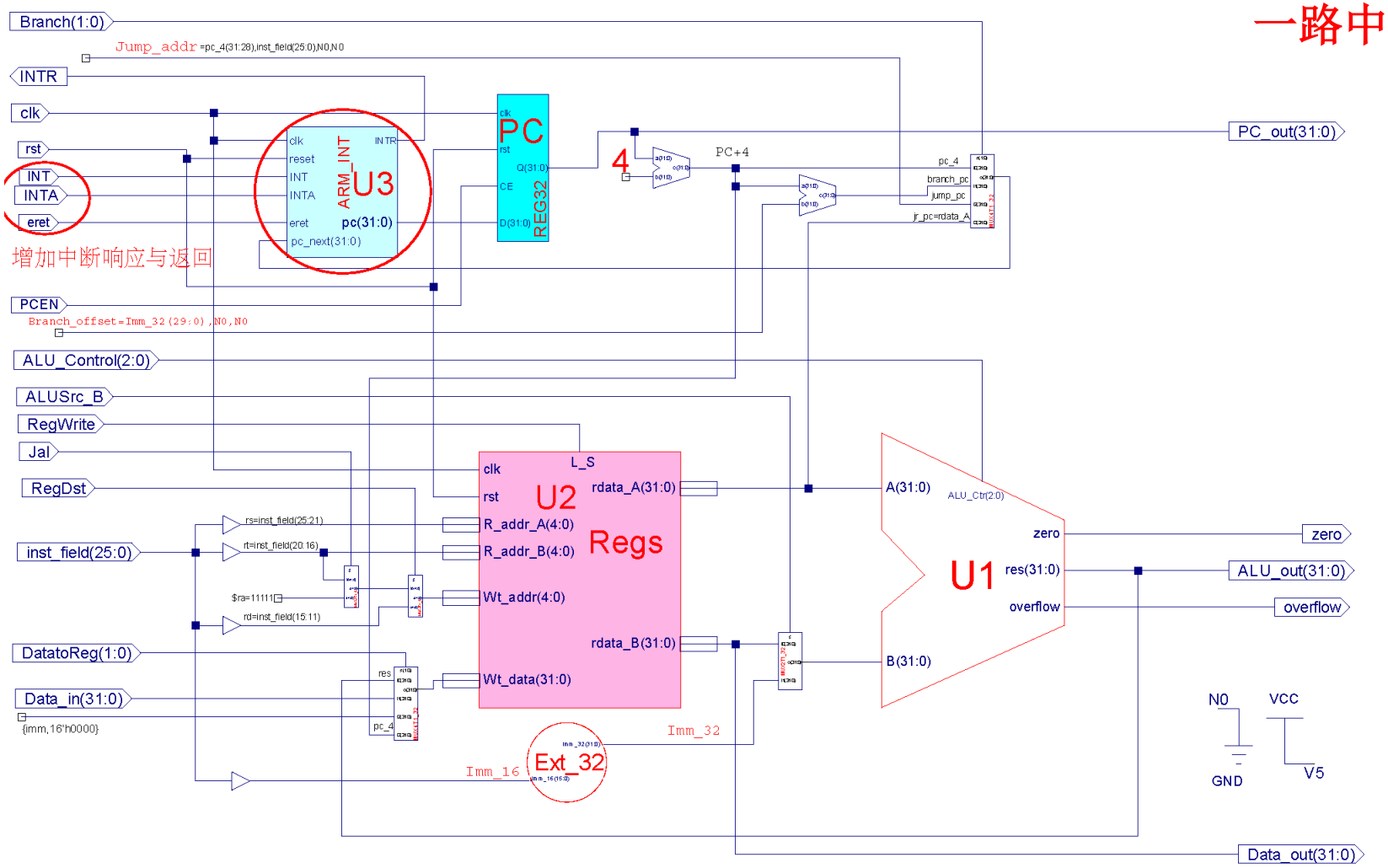
控制器扩展中断译码

- **控制器修改**
  - ARM模式(简单)
    - 仅增加eret指令
    - 中断请求信号触发PC转向，在Datapath模块中修改
  - MIPS模式(可独立模块)*
    - 扩展mfc0、mtc0指令译码
    - 增加Wt_Write通道选择控制
    - 增加CP0_Write
    - 增加控制信号eret、RI、CP0_Write
- **中断调试**
  - 首先时序仿真
  - 物理验证
    - 用BTN[0]触发调试：静态或低速
    - 用计数器counter1_OUT调试：动态或高速

      注意动态测试时死锁!!!

ARM中断模式的DataPath

一路中断



ARM interrupt description

- 中断触发检测与服务锁存

检测与锁存

```
assign INTR = int_act;
assign int_clr = reset| ~int_act;          //clear interrupt Request
always @(posedge clk)
INT_get <= {INT_get[0],INT};               //Interrupt Sampling

always @(posedge clk)begin                 //interrupt Request
   if(INT_get==2'b01)int_req_r<=1;         //set interrupt Request
   else if(int_clr)int_req_r<=0;           //clear interrupt Request
end
```

- 断点保护、中断开、并与返回

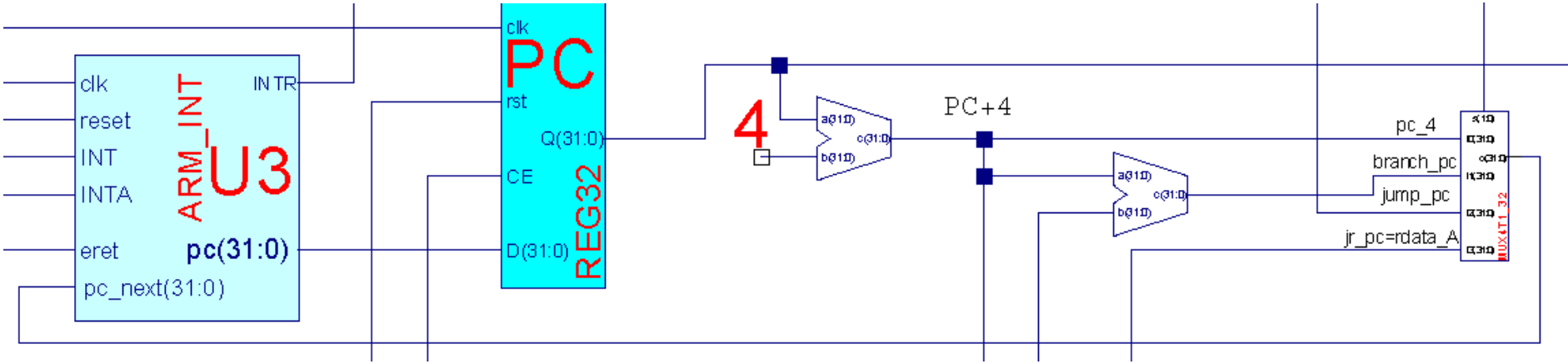
```
always @(posedge clk or posedge reset ) begin
      if (reset)begin EPC     <= 0;          //EPC=32'h00000000;
                      int_act <= 0;
                      int_en  <= 1;
      end
      else if(int_req_r & int_en)begin       //int_req_r: interrupt Request reg
              int_act <= 1;                  //interrupt Service set
              int_en  <= 0;                  //interrupt disable
           end
           else begin if(INTA & int_act)  begin
                        int_act<=0;
                        EPC <= pc_next;   end   //interrupt return PC
                      if(eret) int_en<=1;       //interrupt enable if pc<=EPC;
                  end
end
```

中断通路模块：PC通路

- PC输出通路

仅实现一路中断，请同学们扩展

```
always @* begin
    if (reset==1)pc <= 32'h00000000;
    else if(INTA)
            pc <= 32'h00000004;          //interrupt Vector
        else if(eret)pc <= EPC;          //interrupt return
            else pc <= pc_next;          //next instruction
end
```

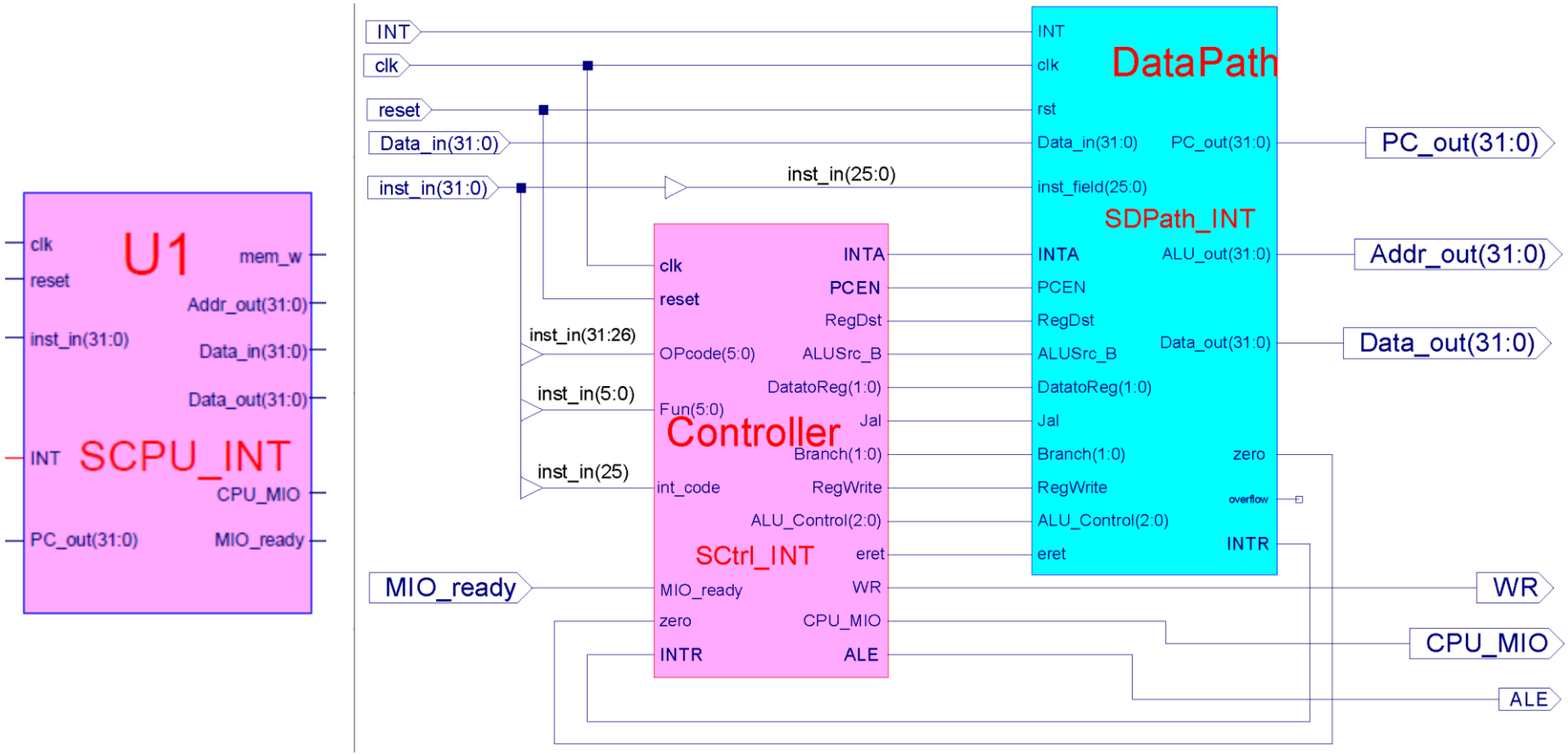


- SCtrl_INT ?

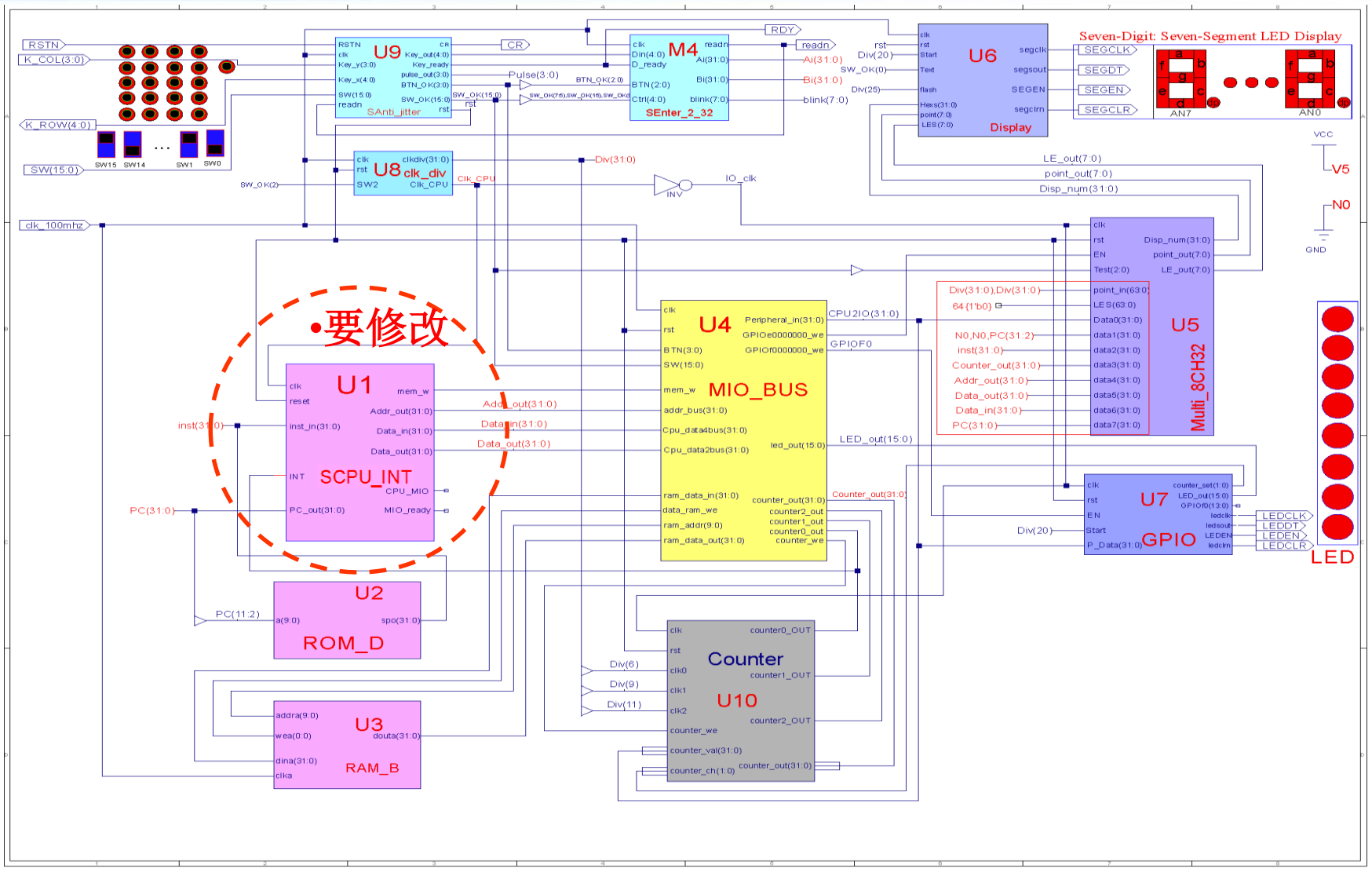
INTA ⟷ INTR

增加中断后的CPU模块

- 注意修改模块逻辑符号
  - SCPU_INT.sym、SCPU_ctrl_INT.sym和Data_path_INT.sym



SOC作为调试测试环境



修改功能测试程序：判断子代码

```
loop1:
      lw  $a1, 0($v1);        //读GPIO端口F0000000状态，同前。
      add $a1, $a1, $a1;
      add $a1, $a1, $a1;      //左移2位将SW与LED对齐
      sw  $a1, 0($v1);        //再将新$a1:r5写到GPIO端口F0000000,写到LED
      lw  $a1, 0($v1);        //再读GPIO端口F0000000状态
      and $t3,$a1,$t0;        //与80000000相与，即：取最高位=out0,屏蔽其余位
      beq  $t3,$t0,C_init;  //硬件计数。out0=1,Counter通道0溢出,中断转C_init
      add $t5, $t5, $v0;    //程序计数延时(加1)
     // beq $t5, $zero,C_init;  //若程序计数$t5:r13=0,转C_init
l_next:                          // 延时未到，继续：判断7段码显示模式：SW[4:3]
      lw  $a1, 0($v1);           //再读GPIO端口F0000000开关SW
      add $s2, $t6, $t6;         //因$t6:r14=4, 故$s2:r18=00000008
      add $s6, $s2, $s2;         //$s6:r22=00000010
      add $s2, $s2, $s6;         //$s2:r18=00000018(00011000)
      and $t3, $a1, $s2;         //取SW[4:3]到$t3
      beq $t3, $zero, L20;       //SW[4:3]=00,7段显示"点"左移反复
      beq $t3, $s2, L21;         //SW[4:3]=11，显示七段图形
      add $s2, $t6, $t6;         //$s2:r18=8
      beq $t3, $s2, L22;         //SW[4:3]=01,七段显示预置数字
      ……
```

三种定时选择

•SW状态循环显示

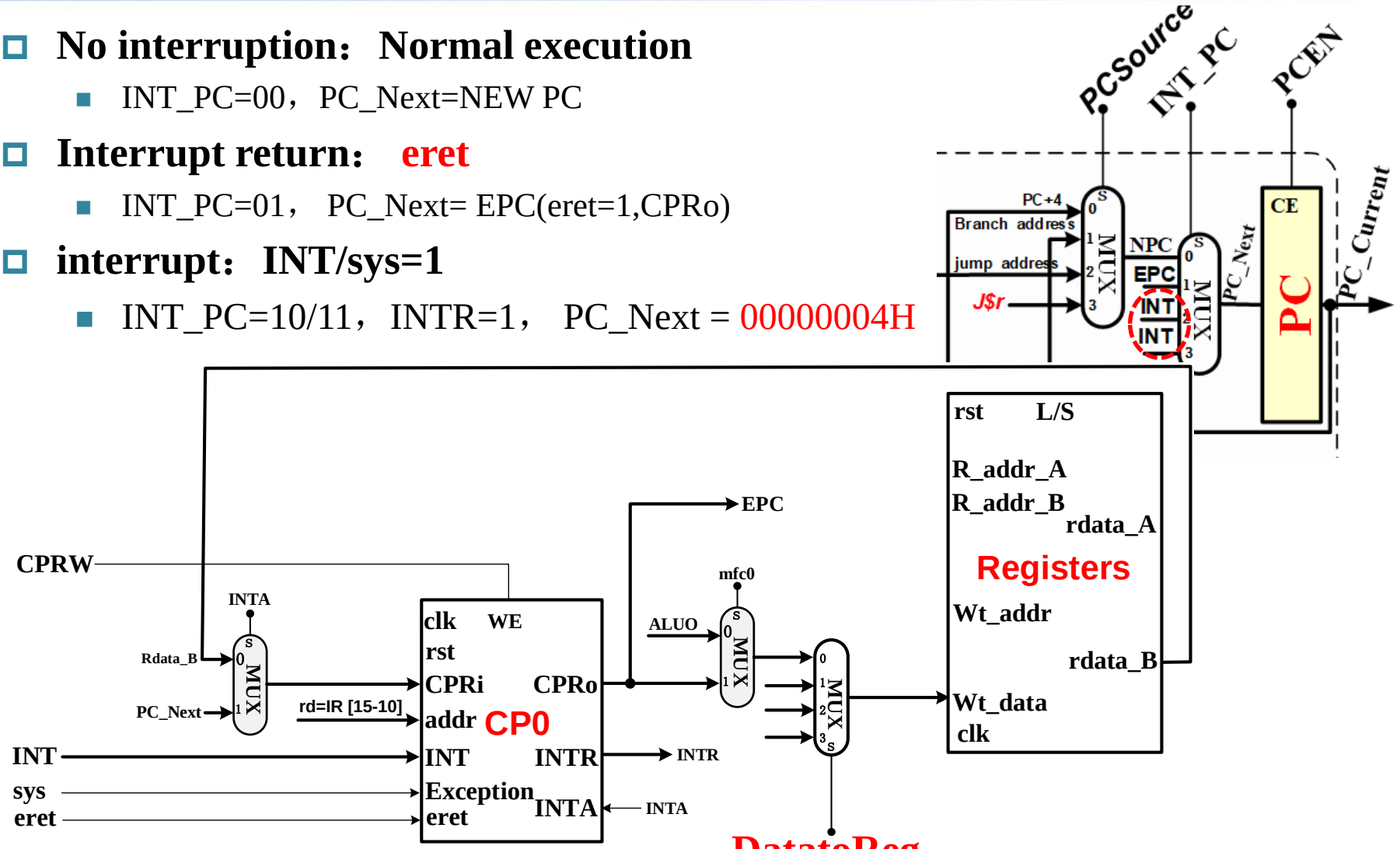
修改定时子代码：中断返回

```
C_init:                    //延时结束，修改显示值和定时/延时初始化
    lw  $t5, 14($zero);    //取程序计数延时初始化常数
     add $t2, $t2, $t2;     //$t2:r10=fffffffc，7段图形点左移
     or  $t2, $t2, $v0;     //$t2:r10末位置1，对应右上角不显示
     add $s1, $s1, $t6;     //$t6:r14=00000004，LED图形访存字地址加1
     and $s1, $s1, $s4;     //与3F相与，留下后6位。$s1:r17=000000XX，//屏蔽地址高位
     add $t1, $t1, $v0;     //r9+1
     beq $t1, $at, L6;      //若r9=ffffffff,重置r9=5
     j L7;
L6:
     add $t1, $zero, $t6;   //r9=4
     add $t1, $t1, $v0;     //重置r9=5
L7:
     lw  $a1, 0($v1);       //读GPIO端口F0000000状态
     add $t3, $a1, $a1;
     add $t3, $t3, $t3;     //左移2位将SW与LED对齐，同时
     sw  $t3, 0($v1);       //r5输出到GPIO端口F0000000
     sw $a2,4($v1);         // 计数器端口:F0000004，送计数常数
     eret;
```

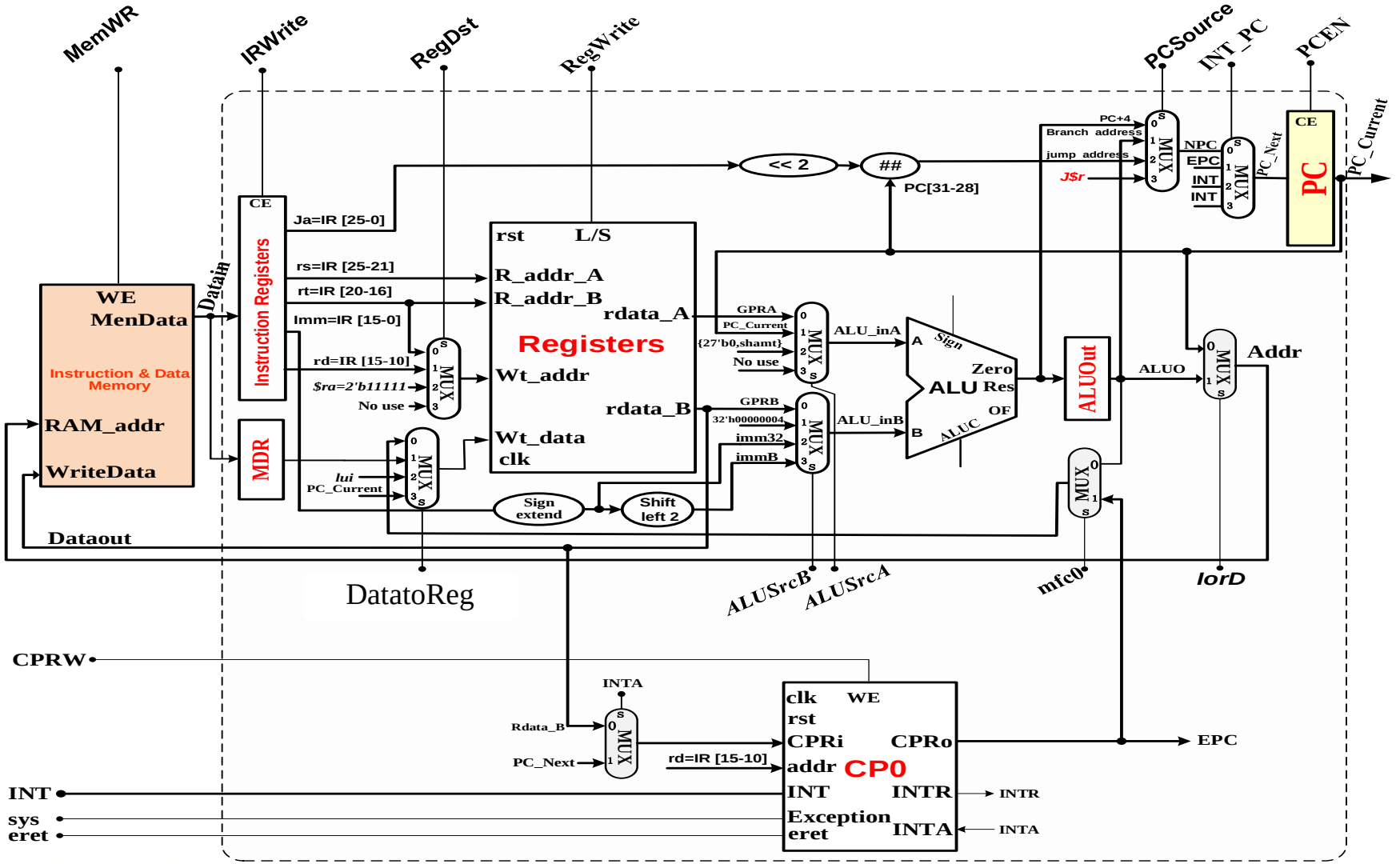
注意动态测试时死锁!!!

CP0 Datapath

- **No interruption：Normal execution**
  - INT_PC=00，PC_Next=NEW PC
- **Interrupt return： eret**
  - INT_PC=01， PC_Next= EPC(eret=1,CPRo)
- **interrupt：INT/sys=1**
  - INT_PC=10/11，INTR=1， PC_Next = 00000004H



Interruption Implementation



CP0 Interrupt Description

```
1  module      CP0(input  clk,rst,
2                  input  INTA,
3                  input  [4:0] addr,
4                  input  [31:0]CPRi,
5                  input  WE,
6                  input  [7:0]INT,
7                  input  Exception,
8                  input  eret,
9                  output [31:0] CPRo,
10                 output [31:0] Cause,
11                 output [31:0] Status,
12                 output INTR );
13 reg [31:0] CP0 [0:3];                        // CP0.0-CP0.3
14 reg [7:0]Trig;
15 reg [1:0] req_get;
16 reg int_clr,int_act;
17 integer i;
18
19    wire [4:0] Exc_code = {1'b0,Exception,3'b000};
20    assign EPC  = CP0[2];
21    assign Cause   = CP0[1];
22    assign Status  = CP0[0];
23    assign int_en  = Status[0];
24    assign INTR = int_act;
25    wire [7:0] INT_MK   = INT & Status[15:8];          //TInterrupt mask
26    assign int_req = | {INT_MK, Exception };
27    always @(posedge clk)
28       req_get <= {req_get[0], int_req};               //外部中断采样
```

ExcCode：sys=08=5’b01000

CP0 Interrupt Description-1

```
29  //interrupt Trigger
30     always @(posedge clk)
31        if(rst)
32           int_act <= 0;
33        else if(int_en && (req_get=2'b01))
34                 int_act <= 1;                                  //*interrupt Service
35              else if(INTA & int_act)                           // ACK interrupt Service
36                  int_act<=0;                                   // clear interrupt Request
37
38  // CP0 Register Access
39     assign CPRo =(~eret) ? CP0[{addr[1:0]}] : EPC ;            // read
40
41     always @(posedge clk or posedge rst) begin
42           if (rst==1) begin                                    // reset
43           int_clr <=0 ;
44           CP0[0] <= 32'h0000FF00;
45           for (i=1; i<4; i=i+1)CP0[i] <= 0;                    //CP0 clear;
46        end
47        else begin
48             if ((addr[4:2] == 3'b011) &&(WE == 1))            // write
49                 CP0[{addr[1:0]}] <= CPRi;
50              else begin
51                 if(INTA | Exception)begin
52                 CP0[2]  <= CPRi;                          //Save interrupt return PC
53                 CP0[1]  <= {Cause[31:16],INT_MK,1'b0,Exc_code,2'b00};
54                 CP0[0]  <= {Status[31:1], 1'b0};
55                 end
56                 if(eret) CP0[0]  <= {Status[31:1], 1'b1};    //restore interrupt enable if pc<=EPC
57              end
58        end
59     end
60  endmodule
```