PCMux

Input

```
- PCAdd: PCAdd = PC + imm  -- B类指令
- PCRx : PCRx  = rx        -- JR指令
- PCplus1: PCplus1 = PC + 1 -- PC正常加一
- PCMuxSel: PC选择信号 TYPE SEL_PC IS (PCADD, PCRX,
PCPLUS1);
```

Output

```
- NewPC:
```

PC

Input

```
- NewPC: 新的PC值
- Keep:  是否保持PC而不采用新的PC值
```

Output

```
- PC: PC = NewPC
```

Adder

Input

```
- PC: PC
- 1
```

Output

```
- PCplus1: PCplus1 = PC + 1
```

InsMemory -- Ram2

Input

```
- PC: STD_LOGIC_VECTOR(15 downto 0);  -- Instruction
Address
- Ram2Data: STD_LOGIC_VECTOR(15 downto 0); -- Instruction
from Ram2
```

Output

```
- Ram2Addr: STD_LOGIC_VECTOR(15 downto 0);
- Ram2CE: Ram2 控制信号
- Ram2OE: Ram2 控制信号
- Ram2WE: Ram2 控制信号
- Instruction: STD_LOGIC_VECTOR(15 downto 0);
```

MUX_IF_ID

Input

```
- PCplus1: PC + 1 后的值
- Instruction: IF段获取的指令
- Keep: 是否停止一个周期
```

Output

```
- Instruction: STD_LOGIC_VECTOR(15 downto 0)
- PCplus1
- rx: Instruction(10 downto 8)
- ry: Instruction(7 downto 5)
- Imm(11): Instruction(10 downto 0);
- rz: Instruction(4 downto 2)
```

ControlUnit: Decoder

Input

```
- Instruction:
- Condition: 来源为data1, BEQZ, BNEZ, BTEQZ, BTNEZ用来比较的数, 根据比较结果来决定 PCMuxSel 的值
```

Output

- WB 阶段控制信号
  - DstReg: WB阶段的目的寄存器
  - RegWE: WB阶段的写使能
- MEM 阶段控制信号
  - MemRead: 是否读数据, WB阶段的数据选择来源， ALUOut 还是 MemDout 的数据。若读，则为 MemDout, 否则为 ALUOut
  - MemWE: 是否写内存
- EXE 阶段控制信号
  - ALUop: ALU 的操作类型
  - ASrc: ALU 前面的 A 数据选择器选择信号
  - BSrc: ALU 前面的 B 数据选择器选择信号
- ID 阶段控制信号
  - PCMuxSel:
  - ImmeSrc: 包括11位（B指令），8位，5位，4位，3位
  - ZeroExtend: 立即数是否为0扩展

RegisterFile

Input

```
- PCplus1:向后传递PC，MFPC 指令需要 PC 的值
- Read1Register: rx
- Read2Register: ry
- WriteRegister: WB.DstReg，WB 阶段的目标寄存器
- WriteData: WB.DstVal，WB 阶段要写入目标寄存器的数值
- Data1Src: Data1的来源，TYPE DataSrc IS (None, Rx, Ry,
PCplus1, SP, T, IH)
- Data2Src: Data2的来源, DataSrc
- RegWE: WB阶段的寄存器的写使能
```

Signals

```
- TYPE Register IS STD_LOGIC_VECTOR(15 downto 0);
- R(0-7);
- SP, T, IH;
```

Output

```
    JR 指令的PC需要选择 rx
    BEQZ, BNEZ 需要比较R[x]的值是否为0或非0，然后给出PCMuxSel信
号
    BTEQZ, BTNEZ 需要比较T的值。
    用 Data1Src 控制信号来统一比较Data1的值。
```

```
- Data1: Output数据1
- Data2: Output数据2
```

MUX_ID_EXE

Input

```
- rx:        源寄存器 rx
- ry:        源寄存器 ry
- data1:     RegisterFile 输出的数据 1
- data2:     RegisterFile 输出的数据 2
- Immediate: STD_LOGIC_VECTOR(15 downto 0);
- DstReg:    目的寄存器
- RegWE:     寄存器写使能
- MemRead:   内存读, 只有 LW 和 LW_SP 两条指令时为 1,用于
HazardDetectingUnit
- MemWE:     内存写使能
- ALUop:     ALU 操作
- ASrc:      ALU A 选择器的选择信号
- BSrc:      ALU B 选择器的选择信号
- Stall: 气泡，暂停信号
```

Output

```
- data1:
- data2:
- Immediate: 立即数
- MemWriteData: 要写入内存的寄存器中的值, 用于 SW, SW_SP 指令
- rx: 源寄存器1，给 ForwardingUnit 的
- ry: 源寄存器2，给 ForwardingUnit 的
- DstReg:    目标寄存器: R0 ~ R7, SP, T, IH
- RegWE:     寄存器写使能
- MemRead:   内存读
- MemWE:     内存写使能
- ALUop:     ALU 操作
```

```
- ASrc:      ALU A 选择器的选择信号
- BSrc:      ALU B 选择器的选择信号
```

ImmExtend

Input

```
- ImmeSrc: 立即数位数及来源， 包括11位（B指令），8位，5位，4位，3位
- ZeroExtend: 是否为0扩展，ZeroExtend为0时采用符号扩展。只有LI指令为ZeroExtend。
- Imme(11): Instruction(10 downto 0);
```

Output

```
- Imme(16):
```

MUX_ALU_A 四选一数据选择器

Input

```
- data1
- imm
- ExE/MEM.ALUOut
- MEM/WB.DstVal
- ASrc
- ForwardingA: STD_LOGIC_VECTOR(1 downto 0)   -- 是否旁路，如果需要旁路，选择 EXE/MEM 的还是 MEM/WB
```

Output

```
- opA
```

MUX_ALU_B 四选一数据选择器

Input

```
- data2
- imm
- EXE/MEM.ALUOut
- MEM/WB.DstVal
- BSrc
- ForwardingB: STD_LOGIC_VECTOR(1 downto 0)   -- 是否旁路，如果需要旁路，选择 EXE/MEM 的还是 MEM/WB
```

Output

```
- opB
```

ALU

Input

```
- opA:  STD_LOGIC_VECTOR(15 downto 0)
- opB:  STD_LOGIC_VECTOR(15 downto 0)
- ALUop: TYPE IS (
    OP_NONE,-- No operation
    OP_ADD, -- F <= A +  B
    OP_SUB, -- F <= A -  B
    OP_AND, -- F <= A &  B
    OP_OR,  -- F <= A |  B
    OP_CMP, -- F <= A != B, not equal
    OP_LT,  -- F <= A <  B
    OP_POS, -- F <= A
    OP_SLL, -- F <= A << B
    OP_SRL, -- F <= A >> B(logical)
    OP_SRA, -- F <= A >> B(arith)
    );
```

Output

```
- F: ALUOut
- T: 标志位，包括加减法溢出，结果为0等。需要多个标志位
```

MUX_EXE_MEM

Input

```
- DstReg:
- RegWE:
- MemRead:
```

```
- MemWE:
- MemWriteData:
- ALUOut:
- T:
- Stall:
```

Output

```
- DstReg:  目标寄存器 R0 ~ R7, SP, T, IH
- RegWE:
- MemWE:
- MemWriteData:
- ALUOut:
```

DataMemory

Input

```
- Address:
- MemRead:
- MemWE:
- WriteData:
```

Output

```
- Ram1CE:   OUT   STD_LOGIC;
- Ram1OE:   OUT   STD_LOGIC;
- Ram1WE:   OUT   STD_LOGIC;
- Ram1Addr: OUT   STD_LOGIC_VECTOR(17 downto 0); -- 需要检测是否为串口地址 0xBF00, 0xBF01等
- Ram1Data: INOUT STD_LOGIC_VECTOR(15 downto 0);
- VGA r,g,b: VGA接口的缓存区
- DataOut:  从 Ram1Data 读出的数据
```

MUX_MEM_WB

Input

```
- ALUOut:  ALU 的计算结果
- MemData: Ram1 读出来的数据
- MemRead: 用于选择 ALUOut 和 MemData
- DstReg:  目标寄存器 R0 ~ R7, SP, T, IH
- RegWE:   是否写目标寄存器
```

Output

- DstReg: WB 阶段的目标寄存器
- RegWE: 是否写目标寄存器
- DestVal: 选择出来的要写入寄存器的值

HazardDetectingUnit

Input

```
""" 检测条件
    1. 上一条指令是 LW 或 LW_SP
    2. 且它的写入寄存器和当前指令的某一源寄存器相同

    ID/EX.MemRead AND
    (ID/EX.DstReg = IF/ID.rx OR ID/EX.DstReg = IF/ID.ry )
"""
- ID/EXE.MemRead: STD_LOGIC; -- 只有 LW 和 LW_SP 指令时为 1
- ID/EXE.DstReg:  STD_LOGIC_VECTOR(3 downto 0);
- IF/ID.rx: STD_LOGIC_VECTOR(2 downto 0);
- IF/ID.ry: STD_LOGIC_VECTOR(2 downto 0);
```

Output

```
"""
让当前指令的控制信号全部为0,即不进行任何写入操作
让PC值保持不变 让IF/ID段寄存器保持不变
"""
- PC_Keep: PC 保持不变
- IFID_Keep: MUX_IF_ID 保持不变
- IDEX_Stall: STD_LOGIC, 暂停信号
```

ForwardingUnit

Input

```
"""
    EXE 段检测条件:
        EXE/MEM.RegWE AND EXE/MEM.DstReg != 0 AND
        EXE/MEM.DstReg = ID/EXE.rx (ry)
    MEM 段检测条件:
        MEM/WB.RegWE AND MEM/WB.DstReg != 0 AND
        (MEM/WB.DstReg = ID/EXE.rx OR MEM/WB.DstReg =
ID/EXE.ry)
"""
- exememRegWE:  STD_LOGIC;
- exememDstReg: STD_LOGIC_VECTOR(2 downto 0);  -- 用于检测
EXE 段数据冲突

- memwbRegWE:   STD_LOGIC;
- memwbDstReg:  STD_LOGIC_VECTOR(2 downto 0);  -- 用于检测
MEM 段数据冲突

- idexeRx:      STD_LOGIC_VECTOR(2 downto 0);
- idexeRy:      STD_LOGIC_VECTOR(2 downto 0);
```

Output

```
- ForwardingA: 可选的值有 ( NotForwarding, EXEForwarding,
MEMForwarding )
- ForwardingB: 可选的值有 ( NotForwarding, EXEForwarding,
MEMForwarding )
```

结构冲突

地址划分 数据:Ram1, 256 * 1024 * 16 bit, 最大地址:0x40000 指令:Ram2, 256 * 1024 * 16 bit, 最大地址:0x40000

```
系统程序区:0x0000~0x3FFF,16K
用户程序区:0x4000~0x7FFF,16K

系统数据区:0x8000~0xBEFF,16K
用户数据区:0xC000~0xFFFF,16K

串口1数据寄存器:0xBF00
串口1状态寄存器:0xBF01
串口2数据寄存器:0xBF02
串口2状态寄存器:0xBF03
```

数据冲突 RAW - Read After Write

没有WAW: Write After Write 冲突 没有WAR: Write After Read 冲突

```
    ADD R1, R2, R3  ALU->   DM  ->

    SUB R4, R1, R5        -> ALU ->

    AND R6, R1, R7                -> ALU

    OR  R8, R1, R9

    XOR R10,R1, R11
```

ForwardingUnit

1. EXE段数据冲突的检测

当前指令的ID/EXE段和上一条指令的EXE/MEM段

本条指令的源寄存器之一和上一条指令的目的寄存器相同，需要将 rx/ry 保存到ID/EX段
上一条指令需要改写目的寄存器,且不是0寄存器
EXE/MEM.RegWE AND EXE/MEM.DstReg != 0 AND
EXE/MEM.DstReg = ID/EXE.rx (ry)

2. MEM段数据冲突的检测

```
MEM/WB.RegWE AND MEM/WB.DstReg != 0 AND
(MEM/WB.DstReg = ID/EXE.rx OR MEM/WB.DstReg = ID/EXE.ry)
```

冲突检测单元 HazardDetectingUnit

必须暂停一个周期的指令

```
    LW   R2, R1, imm  # R1 = Mem[R2 + sign_extend(imm)]

    SUB  R3, R4, R1   # R1 = R4 - R5

    AND  R1, R3       # R1 = R1 & R2

    OR   R1, R4       # R1 = R1 | R4
```

检测条件:

1. 上一条指令是 LW 或 LW_SP
2. 且它的写入寄存器和当前指令的某一源寄存器相同

```
    ID/EX.MemRead AND
    (ID/EX.DstReg = IF/ID.rx OR ID/EX.DstReg = IF/ID.ry )
```

数据旁路 Forwarding

MEM/WB 寄存器到 EXE/MEM 寄存器后的 ALU 数据选择器

```
(MEM/WB.DstReg = ID/EXE.rx) OR (MEM/WB.DstReg = ID/EXE.ry)
ForwardingA ForwardingB
```

暂停流水线

一旦发生此类冲突 暂停流水线一个时钟 让当前指令的控制信号全部为0,即不进行任何写入操作 让PC值保持不变 让IF/ID段寄存器保持不变 将LW指令的结果通过旁路送到ALUInput端 Forwarding逻辑需要增加:

控制冲突

PC add Imm 移到了 ID 段 目前没有准备做分支预测

异常处理

需要实现 EPC 和 CAUSE

EPC

CAUSE