**"Thus a free and open ISA such as RISC-V… innovate simultaneously…designed for modularity and extensions…modifiable…100-member foundation…" (Hennessy and Patterson, "A new golden age for computer architecture", https://iscaconf.org/isca2018/turing_lecture.html, 2018 Turing Award lecture, ISCA, 2018)**

Required Hardware Review: (includes modules, inputs/outputs, timing, clocking):

edge-triggered clocking
combinational logic vs sequential circuits
registers
register files
ALUs
Multiplexors
Memory

Consider 3 general types of instructions to be supported by the processor core, namely

- memory load/store: *ld, sd*
- arithmetic-logical instructions: *add, sub, and, or*
- conditional branch instruction: *beq*

Recall the functions of the 3 types of instructions are shown below (where 'operation' may refer to *add, sub*, *and, or*):

- R[rd]<= M[ R[rs1] + imm ] , M[ R[rs1] + imm ] <= R[rs2]
- R[rd] <= R[rs1] operation R[rs2]
- if R[rs1] == R[rs2] then PC = PC + {imm,1b'0}

All 3 types of instructions have the following common functions:

- using the PC, fetch the instruction from instruction memory
- increment the PC to point to the next instruction
- read one or two register values ( rs1 , rs2 )
- perform some ALU operation ( op , or '+' for *ld*/*sd* , or '-' for *beq*)

After these common functions, the next operation will differ. For example

- the *ld* will access data memory with ALU output address
- the *sd* will store a register value to data memory at address computed from ALU output
- the ALU instruction will store the ALU output back to a register
- the *beq* instruction will change the value of PC by adding it to an immediate value or not change the PC value depending on the ALU subtraction outcome

- Figures below were revised version of slides associated with the course text (Patterson and Hennessy 2018)

- Hardware implementation will assume :
  - positive edge-triggered clocking.
- Assume the PC and Registers are implemented as positive edge-triggered flipflops with load/CE and reset signals.
- Also assume the memory stores data values also on the positive-edge

General Architecture in showing major components, left to right:

- PC register (along with 2 dedicated adders above it for updating PC value.)

- Instruction Memory
- Registers

  inputs rs1, rs2, rd 5-bit ‘address’ values from instruction
  write data - value to be written to register at address rd if RegWrite==1
  outputs - R[rs1],R[rs2] values stored in registers at addresses rs1,rs2
- ALU - if output of ALU is all ‘0’ then Zero = 1
- Data Memory

Figures 4.1-4.5 show further details, including highlighting flows of instruction types.

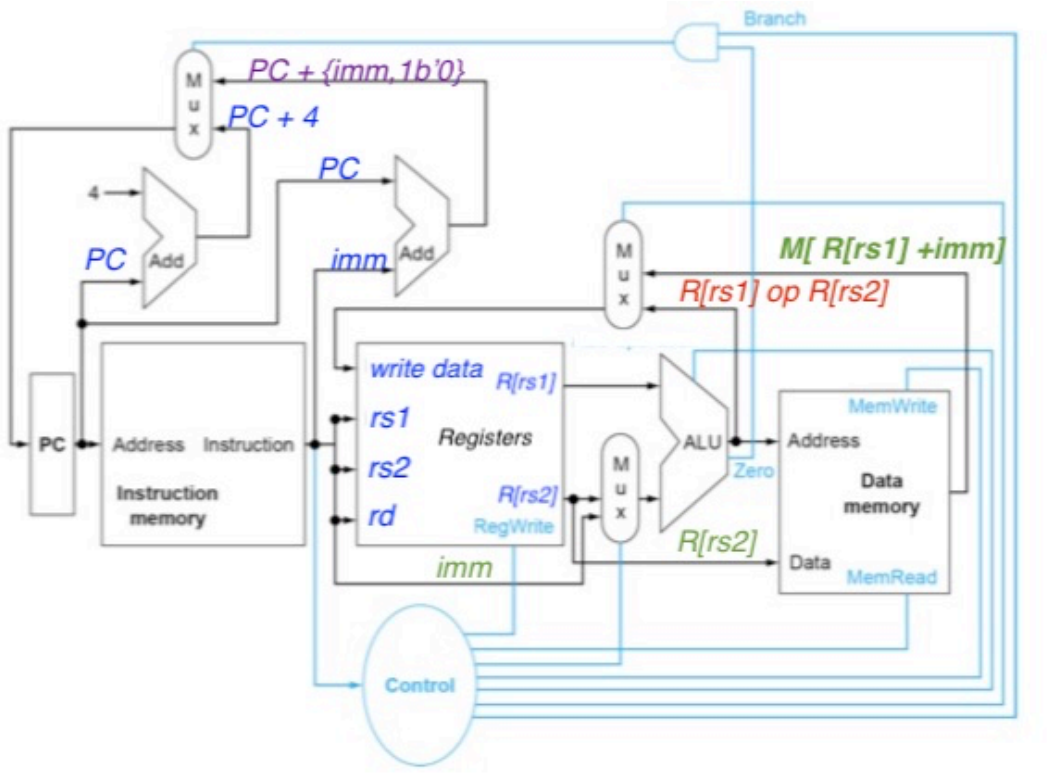



Figure 4.1[1]
General layout of ‘simplistic’ RISCV core

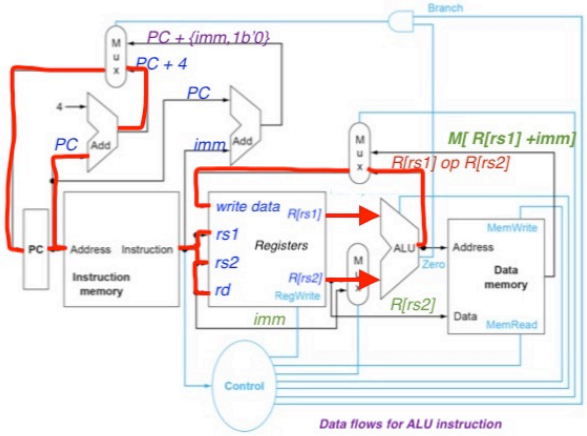



Figure 4.2
RISCV core showing ALU instruction flows

[1] most figures in this chapter are taken from P&H slides, and text, and revised further as shown.

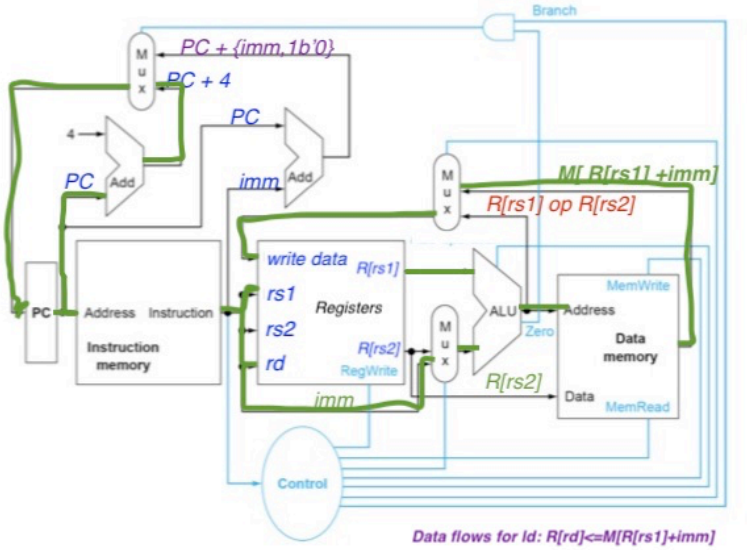



Figure 4.3
RISCV core showing LD instruction flows

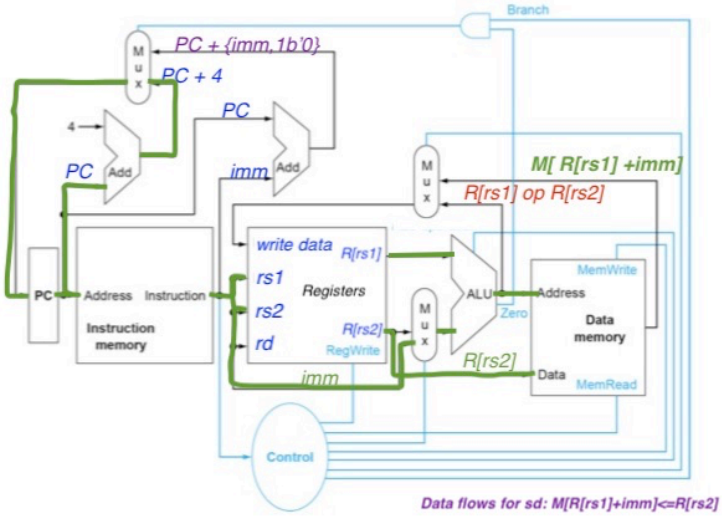



Figure 4.4
RISCV core showing SD instruction flows

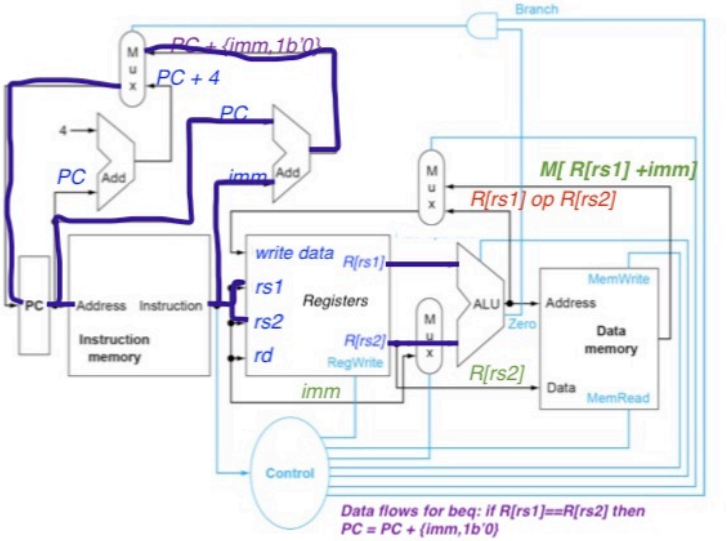
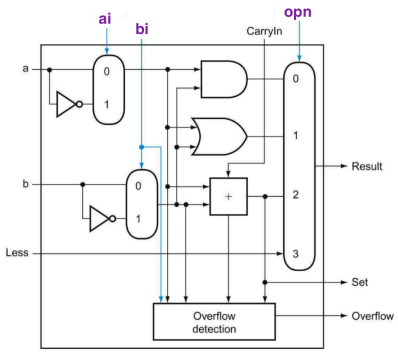



Figure 4.5
RISCV core showing BEQ instruction flows

further details of core:

- *ld,sd* instructions need to sign extend 12-bit immediate to 64-bit signed value (instruction defines imm[11:0] in I-instruction format)
- *beq* instruction needs to concatenate a '0' bit to immediate in the lsb (imm[0]) and sign extend it (instruction defines imm[12:1] in SB-instruction format) to 64-bit signed value. branch taken or not depending upon Zero signal.
- assume all instructions execute in one clock cycle (hence separate instruction and data memory units)
- the control block is split into two blocks: CONTROL and ALU CONTROL
- further details of ALU control signals: 4-bit ALU control signals (will be output from ALU CONTROL block): ai (Ainvert), bi (Binvert), opn(operation). see appendix A for details and table below. (ai and bi invert the input data if they are asserted. opn selects the output from and, or, add/sub, set.)




| ai | bi | opn | alu func | details |
|---|---|---|---|---|
| 0 | 0 | 0 0 | a & b | and |
| 0 | 0 | 0 1 | a \| b | or |
| 0 | 0 | 1 0 | a + b | add |
| 0 | 1 | 1 0 | a + (-b) | sub |
| 0 | 1 | 1 1 | a + (-b) | set if less than , result <= "0…0" \|\| msb( a + (-b) ) |

Note that the Opcode field of each instruction completely determines the following control signals, where ALUOp is added and will be input to the ALU control logic which outputs the above ALU control input signals (all set to 0 unless otherwise stated below).

- R-format instruction : RegWrite=1, ALUop=10
- I-format instruction : ALUSrc=MemtoReg=RegWrite=MemRead=1
- S-format instruction : ALUSrc=MemWrite=1
- SB-format instruction : Branch=1 ALUop=01

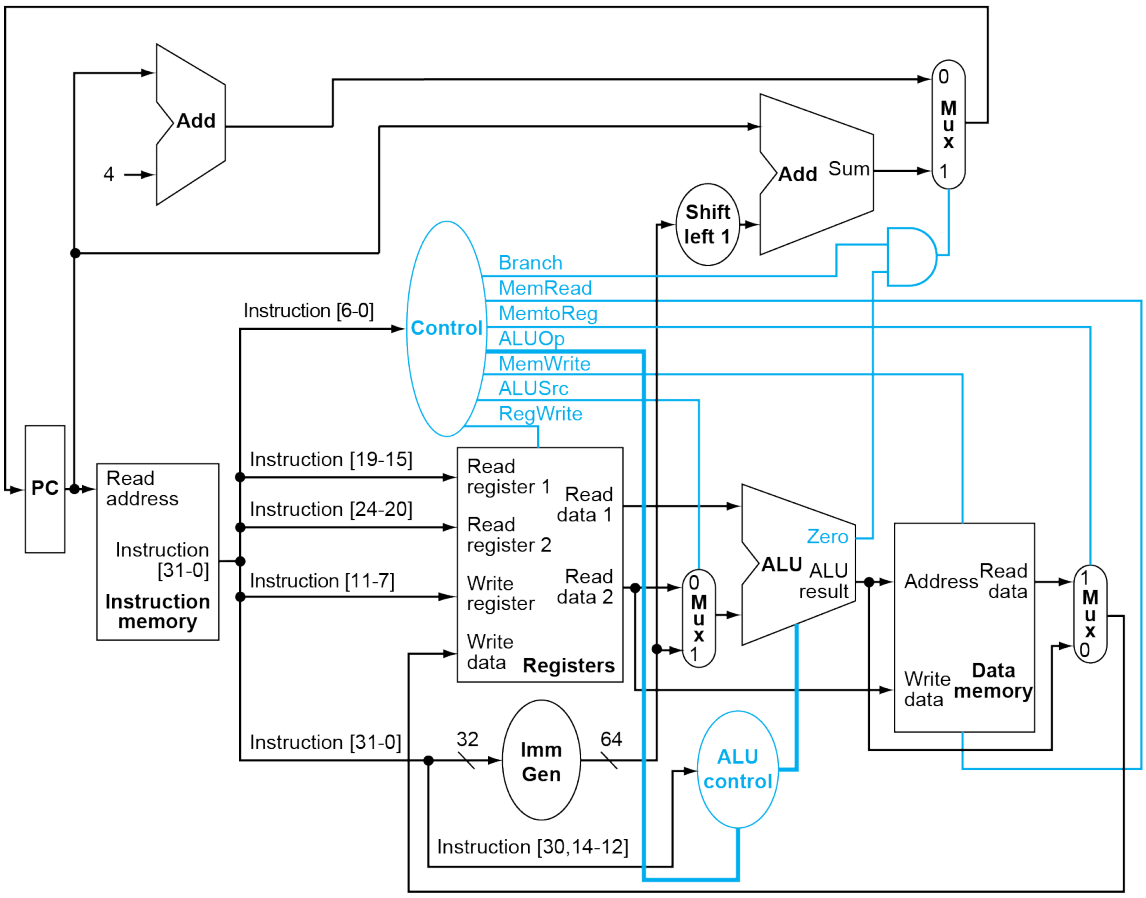
Thus these signals are the 8-bit output from the CONTROL block (whose inputs are strictly the opcode field of the instructions, instruction[6:0])
CONTROL block 8-bit output signals:

- execution signals (EX): ALUop, ALUSrc
- memory access signals (M): Branch, MemRead, MemWrite
- write-back signals (WB): RegWrite, MemtoReg

The ALUop (output from CONTROL block) is input to the ALU CONTROL block along with function field 3 and 2nd msb of function field 7, specifically instruction[30] to differentiate add from subtract functionality. The ALU control signals are output from ALU CONTROL block.

| instruction | ALUOp | functn7 instruction[30] | functn3 | ALU function | ALU control signals (ai,bi,opn) |
|---|---|---|---|---|---|
| ld | 0 0 | x | xxx | add | 0 0 1 0 |
| sd | 0 0 | x | xxx | add | 0 0 1 0 |
| beq | 0 1 | x | xxx | subtract | 0 1 1 0 |
| add | 1 0 | 0 | 0 0 0 | add | 0 0 1 0 |
| sub | 1 0 | 1 | 0 0 0 | subtract | 0 1 1 0 |
| and | 1 0 | 0 | 1 1 1 | and | 0 0 0 0 |
| or | 1 0 | 0 | 1 1 0 | or | 0 0 0 1 |

Split control blocks are shown below for final processor core design:



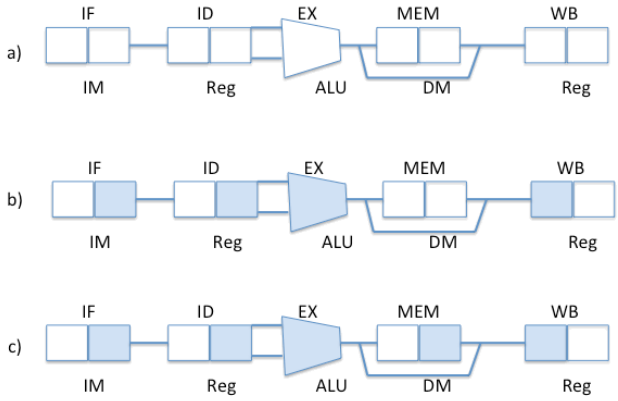

This is a single-cycle processor implementation. The critical path is likely the load instruction, requiring register file access, ALU address calculation, memory read and register file write.

Hence performance will be poor (limited by slowest instruction). In the next section this processor is divided into 5 parts (instruction memory part, register reading part, ALU, data memory part and register writing part) to support pipelining and improved performance.

**Pipelining** provides a method of improving the performance significantly.
Consider a 5-stage pipeline

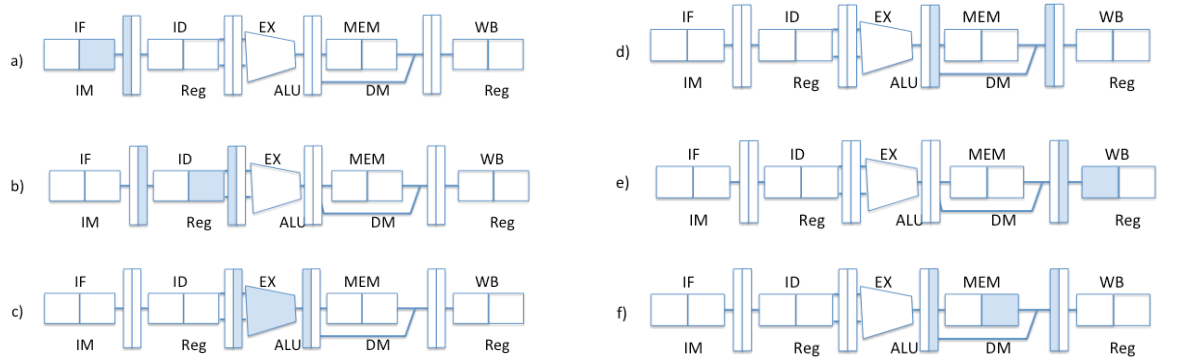
1. IF: instruction fetch
2. ID: instruction decode & register file read
3. EX: ALU execution & address calculation : ALUOp, ALUSrc (EX)
4. MEM: Memory access : Branch, MemRead, MemWrite (M)
5. WB: write-back : MemtoReg, RegWrite (WB)

The original processor designed in the last section can be divided into 5 stages, shown using the flow in a) below, where the top labels (IF, ID, Ex, Mem, WB) denote the name of the stage and the notation under each symbol represents the hardware involved . The IF is using the instruction memory (IM), ID is reading register values from register file (Reg), Ex is the flow through the ALU, MEM is the possible load or store of data in the data memory (DM) and WB is writing data into the register file (Reg).



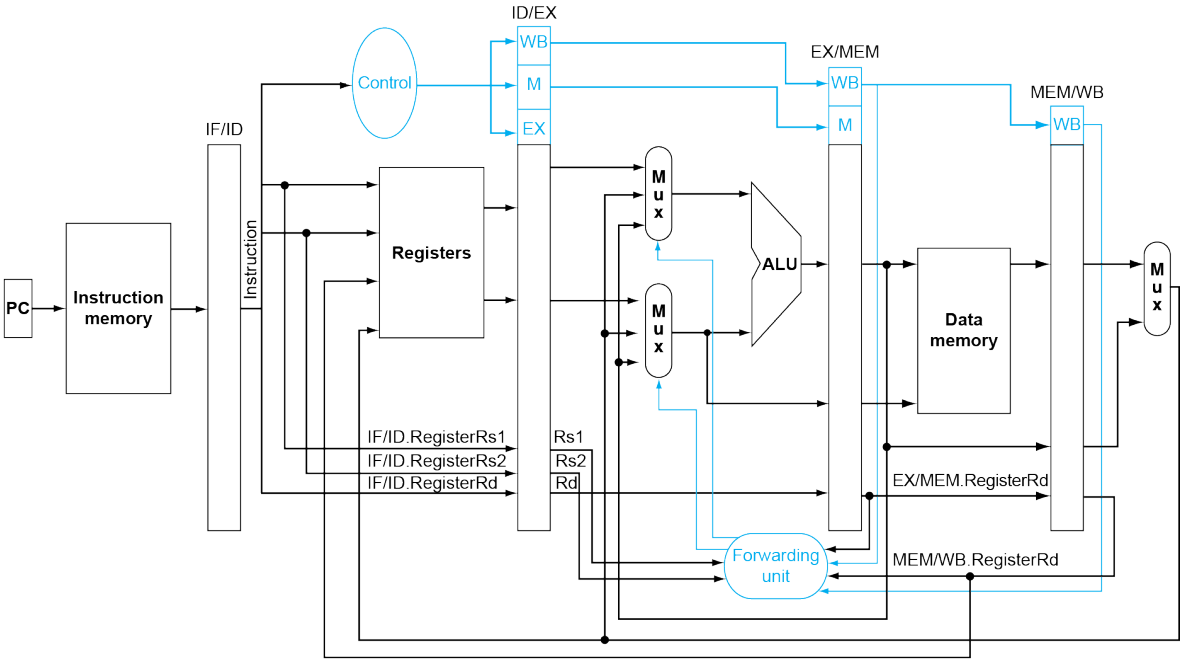

The shading indicates what part of the flow is active for any instruction. For example a R-type instruction, such as *add x5,x7,x6*, would be illustrated with shading in b) on left. Assume the register file is written during the first half of the clock cycle and read during the 2nd half of the clock cycle (note falling edge of clock is the active edge for the register file, whereas all other state elements use the rising edge of clock). The shaded portion on the right of a state element (IM,Reg,DM) indicates a read of a value, and the shading on the left indicates writing a value. The shading of the ALU indicates it is active during that instruction. The flow in c) represents the active flow for a load word, where ALU computes address for DM where data is read out and then written into register file in WB stage.

To now understand pipelining, a (pipeline) register is placed in-between each of the stages (IF, ID, EX, MEM, WB) so that during one clock cycle at most all 5 stages may be active performing different instructions, see a) through f) below. If left half of pipeline register is shaded it is being written, whereas if right half is shaded it is being read. Note that the ID and WB stages are simultaneously using the register file (Reg), however the ID is reading during the 2nd cycle of the clock and the WB is writing during the first cycle of the clock cycle (thus an instruction can write back a value and in the same clock cycle read it for another instruction). For example, a)



through e) represent flow for *add* instruction, whereas a)-c),f),e) represents flow for *lw* instruction respectively.

The more detailed data path is shown below with the 5 pipestages. Between each stage are the pipeline registers which now include forwarding of control signals. For example after the first IF stage the pipeline register is denoted by IF/ID since on the edge of the clock the instruction fetched data is stored into the pipeline register and starts to then propagate through the instruction decode path. The next pipeline register contains WB, M (Mem) and EX control signals which will be forwarded to those appropriate stages.




The control signals (EX, M, WB) must be placed into pipeline registers such that they're ready to be used in the appropriate pipe stage (EX is stored in one pipeline register, M is stored in two pipeline registers and WB needs to be stored in 3 pipeline registers)

Note 1. that there are timing issues or **data hazards** with this pipeline, for example consider instruction I0 fetched in clock cycle 0, followed by instructions I1 and I2:

I0: **x1**<= x2 + x3
I1: x5<= **x1** + x4
I2: x6<= **x1** + x5

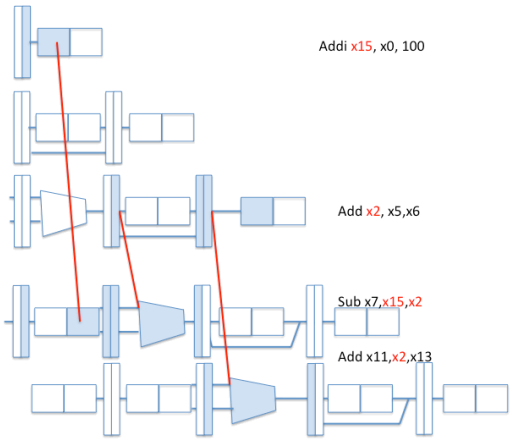
register *x1* of I0 is not written back to the register file until clock cycle 4, yet *x1* should be fetched in clock cycle 2 and input to ALU in clock cycle 3. The value of register *x1* is produced in clock cycle 2 by I0, hence it can be Forwarded to the ALU in clock cycle 3 for I1.
Similarly *x1* will also need to be forwarded to ALU in clock cycle 4 for use by I2.

Note 2. there are other timing issues or **data hazards**. for example consider the following code sequence, where again instruction I0 is fetched in clock cycle 0:

I0: ld **x1**, 20(x2)
I1: x5<= **x1** + x4

*x1* is not available until read from memory in cycle 3, but is needed as input to ALU during the same cycle 3. this is not possible, hence one must introduce a *nop* after the load or **stall**. This may be avoided if the compiler can reorder statements to avoid a stall.

Diagram below is an example of data forwarding shown with red lines. Consider the code sequence:




addi x15, x0, 100
add x2, x5, x6
sub x7, x15, x2
add x11, x2, x13

The first, second and third clock cycles use forwarding, shown with red lines. During write back stage of *addi*, the data is forwarded to ID stage of *sub* instruction to supply *x15*. During MEM stage of *add x2,x5,x6* the variable *x2* is forwarded to ALU input of *sub*. Finally the write back stage of *add x2,x5,x6* forwards the value of *x2* to the ALU stage of *add x11,x2,x13*.

Note 3. **Control hazards** also exist. consider a *beq* instruction followed by other instructions such as:

I0: beq x1, x0, addr1
I1: add x5, x1, x4

the *beq* outcome is not determined until the output of the ALU, cycle 2, hence the next instruction cannot be fetched until the address is decided in cycle 3 at the earliest, which is a two cycle penalty. By moving the *beq* decision to the instruction decode stage (by XORing register values, instead of using the ALU during the execute stage) the penalty can be reduced by one cycle. typically it is assumed that the branch is not taken, and sequence of instructions are executed until the outcome is known. If the outcome determines that the branch must be taken the pipeline is flushed (by zeroing the instruction field of the IF/ID pipeline registers).

## 4.9 Exceptions and Interrupts

Other than branches, exceptions and interrupts will also change the control of the instruction flow execution.

Exceptions or interrupts may be defined as the unexpected change in control which may be caused by internal or external events respectively.

They were initially designed to handle unexpected events such as an undefined instruction, but today they're also used for I/O device communication (to be discussed in chapter 5).

**Exception** can be defined as unschedule event which disrupts program execution.
**Interrupt** can be described as an exception that comes from outside of the processor.
These terms can be used interchangeably.

Examples follow (only one referred to as interrupt):

- external I/O device interrupt

- system reset
- invoke operating system from user program
- using undefined instruction
- hardware malfunctions

For example, in case of a hardware malfunction or undefined instruction, the processor must save the address of the undefined instruction in the SEPC (supervisor exception cause register) status register and then transfer control to the OS at some specified address.
The cause of the exception is stored in the SCAUSE register.

In general when exception occurs,
- set a register to show the cause of the exception
- the address of offending instruction is saved
- all following instructions are flushed
- all prior instructions are completed

The OS looks at cause of exception and acts accordingly (for hardware failure or undefined instruction it may kill program and return indicator for kill). As will be seen in chapter 5, a page fault may occur which causes an exception.

For I/O device request or OS service call
- saves state of program
- performs task
- later, restores program to continue execution (may have to complete I/O task execution before resuming program)

Since I/O device request and hardware malfunction are not associated with instruction, there's more flexibility in when to interrupt pipeline.

Alternatively in a vectored interrupt method, the address to which control is transferred is determined by the cause of the exception. For example an exception vector address (a specific address defined for undefined instruction or hardware malfunction) is added to a vector table base register. the OS knows what the exception was according to the address it was given. note the SEPC register is still also needed to store the instruction which caused the exception. With device I/O interrupts, one may complete the program before handling the interrupt.

## 4.11 Real Stuff:…

An **IP core** is designed to be incorporated with other logic such as application specific processors (video encoders/decoders etc…), I/O interfaces, and memory interfaces.
Then they are fabricated to yield a processor for a specific application.
Some x86 processors (Intel i7 core) translate the x86 complex ISA into a RISC-V like instruction set, called micro-operations which are then executed in a sophisticated pipelined architecture, or **microarchitecture** (which refers to the detailed internal architecture of a processor, specifically functional units, control, interconnections).

6.9 Communicating to outside world:

In **memory mapped I/O**, portions of the address space are assigned to I/O devices. At boot time, devices can request to be assigned to an address region of a specified length. All subsequent accesses to that region are forwarded to that device. Reads/writes to those addresses are interpreted as commands to that device. When the processor issues the address and data, the memory system ignores it since that region is mapped to I/O. The address encodes the device identity and type of transmission.

However while the processor could transfer data to the device (into the I/O space) from the user space, the overhead for transferring data to an external high speed network would be too high. Thus in these cases a **direct memory access (DMA)** is typically used to offload the processor. DMA provides a device controller with ability to transfer data directly to/from memory without involving the processor.

The DMA sends a I/O interrupt to notify OS that transfer is complete. This **Interrupt-driven I/O** is a scheme that utilizes interrupts to indicate to processor that I/O device needs attention. I/O interrupts have the following characteristics:

1. I/O interrupt is asynchronous with respect to instruction execution (different from overflow, page faults,…). Controller needs only to check for pending I/O interrupt at time it is about to start new instruction.
2. Further information is needed - which device has interrupted, what priority/urgency etc.. this information is communicated by using **vectored interrupts** or an **exception identification register (SCAUSE** as in RISC-V). When processor recognizes interrupt the device sends vector address or a status field to place in the cause register. As a result the OS gets control and can interrogate the device. Interrupt mechanism eliminates need for processor to keep checking on devices.

In Networking application the OS acts as interface between HW and program that request I/O.