Intro to Processor Architecture

Project Report

Y86-64 PROCESSOR

Arshini Govindu : 2020102009

Sankeerthana Venugopal : 2020102008

OVERVIEW

The bare minimum required is to have a working code for the sequential implementation of the Y86-64 architecture design. The goal of this project is to make a full-fledged processor architecture implementation with 5 stage which includes support for eliminating pipeline hazards.

SEQUENTIAL

FETCH

In the fetch stage we read instruction by instruction from the instruction memory and find the values of **icode, ifun , rA , rB** and **valC** according to the instruction. On the positive edge of clock signal the fetch stage gets the updated **PC** value and checks for the instruction in the instruction memory.

If the PC is valid then it takes 10 consecutive bytes from the instruction memory starting from the PC byte and combines it into a single 10 byte i.e. 80 bit register **instr**. The maximum size of a single instruction is 10 bytes. From the instruction register, the **icode** and **ifun** values are extracted by taking the elements **instr**[0:3] and **instr**[4:7] respectively. **valC** is **instr[8:71]** in case of a jump condition. If the **icode** is not among the ones we have written then **instr_valid** becomes 0.

The Sequential folder has **fetch.v** and **fetchtb.v** and that is just to test the fetch stage separately.

DECODE

In the decode stage we output **valA** and **valB** based on the second byte in the instruction along with **icode**. We get **rA** and **rB** from the second byte. We create a register array which represents the register memory. Depending on the **icode,** the values of **valA** and **valB** are calculated. The various registers are then given as input to **valA** and **valB** as required. These operations are represented by **srcA, srcB** and register file blocks. Here, **reg_mem4** represents the %**rsp** register. The values of **valE** or **valM** are written in to **rA, rB** or **rsp** depending on **icode** in the write-back stage. These operations are represented by the **dstE, dstM** and register files.

The Sequential folder has **decode.v** and **decodetb.v** and that is just to test the decode stage separately.

EXECUTE

The execute stage involves the ALU written previously. For some **icode** values, **ifun** is required. And based on that we have to assign values to valE. **cmovXX, jXX** and **OPq** require **ifun** values. For **OPq** we need to use the ALU to perform the required operations. For **cmovXX** we need to check for the move conditions and set **cond**=1 if required.

The Sequential folder has **execute.v** and **executetb.v** and that is just to test the execute stage separately.

MEMORY

The memory stage is used to read and write to memory. According to the value of **icode,** we can decide whether we want to read or write to memory. The data address can be deduced using **icode, valE** and **valA**. The input can be found through **icode, valP** and **valA**. For **rmmovq** we write to memory and for **mrmovq,** we read from the memory.

The Sequential folder has **memory.v** and **memorytb.v** and that is just to test the execute stage separately.

PC UPDATE

This stage is for finding the next value of PC after the whole instruction has been executed. The PC is updated by the instruction length. For most of them, the PC is updated by **valP**. And for **jXX,** it is updated by **valC**.

The Sequential folder has **pc_update.v** and **pc_updatetb.v** and that is just to test the execute stage separately.

PIPELINING

The PC update is the last stage in the sequential implementation but in the pipelined implementation, it needs to be in the beginning as we don't want to have to wait for the new PC at the end of the cycle. There are five stages in the pipelining and they are:

Fetch

The fetch stage of the pipe is similar to that of the sequential one as it is the first stage and all the problems that come with pipelining aren't applicable here yet. The first byte address is the PC value. The remaining 9 bytes of an instruction have information of the register specifier byte and constant. If there is no need for both the registers, they are set to 0 or the sixteenth register specifier. If only one is required then the other is set to 0. The PC update gives **valP**. In the fetch stage we can test for memory error or an illegal instruction address and detect a halt instruction.

Decode

The reg file has **srcA, srcB, dstE** and **dstM**. Each one has an address connectiona and a data connection, where the address is a **regid**. **srcA** indicated which register should be accesed to get **valA**. Register ID **dstE** indicates the destination register for a write port, where the computed value **valE** is stored. But here, the register IDs supplied to the write ports come from the write-back stage

Execute

The execute state includes the ALU. This unit performs the operation add, **subtract, and**, and **xor** operations on the inputs **ip1** and **ip2.** The ALU output **op** becomes **valE** in most cases, otherwise **valE** just takes no value. This stage also includes the setting of condition codes. Our ALU generates the three values on which the condition codes are based—zero, sign, and overflow—every time it operates. However, we only want to set the condition codes when an **OPq** instruction is executed. We use the **carry** output of the ALU to set the condition codes for **ZF, OF** and **SF.**

Memory

This stage sits between the execute and memory stages. It holds the results of the most recently executed instruction for processing by the memory stage. It also holds information about branch conditions and branch targets for processing conditional jumps. It either reads or writes data.

Writeback

This sits between the memory stage and the feedback paths that supply the computed results to the register file for writing and the return address to the PC selection logic when completing a ret instruction.

This processor supports the following instructions:

IHALT	Code for halt instruction
INOP	Code for nop instruction
IRRMOVQ	Code for rrmovq (register to memory) instruction
IIRMOVQ	Code for irmovq (immediate to register) instruction
IRMMOVQ	Code for rmmovq (register to memory) instruction
IMRMOVQ	Code for mrmovq (memory to register) instruction
IOPL	Code for integer operation instructions
IJXX	Code for jump instructions
ICALL	Code for call instruction
IRET	Code for ret instruction
IPUSHQ	Code for pushq instruction
IPOPQ	Code for popq instruction

CHALLENGES FACED

Some of the challenges we faced were understanding Verilog and all the parameters. It took some time to think about the pipeline implementation.