Chapter 4

Overview

- The design principles of simplicity and regularity of MIPS makes the execution of the instructions simpler; every instruction the first two steps are identical:
  - Send the PC to the memory that has the code and fetch that instruction from memory
  - Read one/two register(s) using the field of the instruction to select the registers to read
- All instruction classes, except jump, use the ALU (arithmetic logic unit) after reading the registers
- After the ALU, the instructions begin to differ
  - A memory-reference will need to access the memory to read/write data for a load/store respectively
  - An arithmetic/logical instruction will need to write the data into a register
  - A branch needs to dictate whether or not we change the PC to be something else based on the comparison or incremented by 4 as usual
    - Use the equal output from the ALU for a beq which can be achieved by subtracting the two operands and seeing if it is 0
- In the ALU, there are many places where we have two data sources from which we choose one to finalize the data to send somewhere
  - Use a multiplexer for this
- Several of the units in the CPU are controlled by what instruction it is executing
  - A control unit takes the instruction as an input and determines how to set the control lines

Logic Elements

- The datapath elements consist of two different types of logic elements:
  - Combinational
    - An operational element, such as an AND gate or an ALU
    - Only depends on the current inputs
  - State elements
    - A memory element, such as a register or memory
    - Contains state
      - The state element requires two inputs, the input data and the clock and one output
        - One example is a D-type flip flop
    - The clock is used to determine when the state element should be written
      - A state element can be read at any time

Clocking Methodology

- The clocking methodology is the approach used to determine when data is valid and stable relative to the clock

- Defines when signals can be read and when they can be written
    - Obviously this is necessary as we don't want to read and write at the same time or mix up the order
- MIPS uses an edge-triggered clocking methodology
    - All state changes occur on a clock edge
- All combinational logic elements cannot store state, so its inputs must come from a state element and its output must go into a state element
    - The inputs are values that were written in a previous clock cycle
    - The outputs are values that can be used in a following clock cycle
- If we don't want to change the state on every clock signal, then we can also input a write control signal such that the state element will only change state when the write control is asserted and the clock is asserted
- With an edge-triggered timing methodology, there is no feedback within a single clock-cycle
    - We can read the contents of a register, send the value through some CL, and write to a register in the same clock cycle

Datapath Elements

- Must examine the datapath elements in order to build a datapath
    - A unit used to operate on or hold data within a processor
    - Datapath elements include the instruction and data memories, the register file, the ALU, and the adders

Memory Unit

- The first element we need is a memory unit to store the instructions of a program and supply instructions once given an address
- We have a register called the program counter (PC) that holds the address of the current instruction
    - We fetch an instruction from memory and to prepare for the next instruction, we use an adder to add 4 bytes to the PC to obtain the address of the next sequential instruction

Register File

- The processor's 32 general-purpose registers are stored in a structure called a register file
    - This is a state element that consists of a set of registers that can be read and written by supplying a register number to be accessed
- R Type instructions have three register operands (it reads two registers and writes to one)
    - The regfile must have 5 inputs:
        - 2 Inputs that specify which registers must be read
            - 5 bits each (32 different registers)
            - The contents of these registers are always outputted by the register file constantly
        - 1 Input that specifies the register to write to
            - 5 bits

	- 1 Input that specifies the data being written
	  - 32 bits
	 - 1 Input that specifies if the data should be written or not at the clock edge
	  - Known as the write control signal
	  - Data is only written when both the write control signal and the clock edge is asserted

ALU	- Another datapath element that is necessary is ALU - This is used not only by arithmetic/logical instructions but also by data transfer instructions and branch instructions - Certain instructions that have immediate need to be sign extended, so a sign-extender is another datapath element that is necessary
Data Memory	- For loads and stores, we need a data memory unit as well - The data memory has read and write control signals, an address input, and an input for the data to be written into memory

Branch Calculation	- For branches, we must compute the branch target address relative to the branch instruction address
	 - This is the address specified in a branch, which becomes the new PC if the branch is taken
	 - The base for the branch instruction address is PC+4, which we compute anyway for the sequential case, so we use this as well for the calculation
	 - The offset field must also be shifted left by 2 bits
	- We must also determine whether on not the branch is taken (the branch condition is satisfied and the PC becomes the branch target)
	 - The branch datapath must compare the register contents to determine if the condition is met
	  - To do this, utilize the register file and the ALU
	  - The comparison is indicated by the Zero/Equal signal out of the ALU
	 - It must also actually compute the branch target address
	  - To do this, utilize a sign extension unit, a shifter, and an adder and use the sequential PC calculation as one of the inputs and the sign extended, shifted immediate as the other
	
	- Jumps are branches that are always taken
	- The next PC is calculated by taking the PC and replacing its lower 28 bits with the lower 26 bits of the instruction shifted left by 2 bits

Creating a datapath

- We now have determined all the datapath components needed for each instruction and can combine them into a single datapath with appropriate control signals
- The simplest datapath is a single-cycle CPU (everything will be executed in one clock cycle)
    - Any element needed more than once must be duplicated
        - This leads to separate data and instruction memories
    - To share a datapath element between two different instruction classes, we need to allow multiple connections to the input of an element
        - Use a multiplexor and control signals to select the appropriate signal
- To create a datapath with only a single register file and a single ALU, we must:
    - Support two different sources for the second ALU input
        - For I-type instructions, instead of reading the second input register like R-type, we use the immediate
        - Use a multiplexor with control called *ALUSrc*
    - Support two different sources for the data stored into the register file
        - For load words, we are going to write the data from memory into the specified register and not what comes out of the ALU
        - Use a multiplexor with control called *MemtoReg*
- For branches, we use the ALU already for the comparison test so we must keep the separate adder for computing the target address
    - An additional multiplexor is required to select either the sequentially following instruction address (PC + 4) or the branch target address to be written into the PC

Control Unit

- We need a control unit that dictates the circuit
    - It must be able to take inputs and generate a write signal for each state element, the selector control for each multiplexor, and the ALU control
- Look to the lecture's implementation of the control unit (it seems to be more appropriate)
    - For each instruction being implemented, analyze the RTL (Register Transfer Language) and what control signals are set
    - Create a truth table for the control signals with each instruction as an input
        - A truth table is a representation of a logical operation by listing all the values of the inputs and then in each case showing what the resulting outputs should be
    - Implement boolean logic to determine the control signals given what instruction is being dealt with (known as the "or logic")

	・ We determine what the instruction is by using “and logic” on two input signals - the opcode and the funct, and assert the corresponding output signal for that particular instruction ・ Note we have one output for each possible instruction and these are all fed into the “or logic” and only one of these signals will be set to a high value
Control Signals	・ There are 8 different control signals: ・ RegDst ・ When deasserted, the register destination number for the Write register comes from the rt field (bits 20:16) ・ When asserted, the register destination number for the Write register comes from the rd field (bits 15:11) ・ RegWrite ・ There is no effect when deasserted ・ When asserted, the register on the Write register input is written with the value on the Write data input ・ AluSrc ・ When deasserted, the second ALU operand comes from the second register file output ・ When asserted, the second ALU operand s the sign-extended, lower 16 bits of the instruction ・ PCSrc ・ When deasserted, the PC is replaced by the output of the adder that computes the value of PC + 4 ・ When asserted, the PC is replaced by the output of the adder that computes the branch target ・ MemRead ・ There is no effect when deasserted ・ When asserted, the data memory contents designated by the address input are put on the Read data output ・ MemWrite ・ There is no effect when deasserted ・ When asserted, data memory contents designated by the address input are replaced by the value on the Write data input ・ MemToReg ・ When deasserted, the value fed to the register Write data input comes from the ALU ・ When asserted, The value fed to the register Write data input comes from the data memory ・ ALUOp ・ This determines which operation the ALU will perform on the two inputs ・ Note that the only control that cannot be determined by the decoded instruction is PCSrc, which depends in part on the Equal/Zero signal out of the ALU

	◦ A control signal called Branch can be determined from the control unit, that when placed in an AND gate with the ALU Equal/Zero signal, determines PCSrc • Also note that this doesn't include any control signal we may have for jumps as the book does not address them in it's limited instruction set to model the CPU
Single Cycle Downfalls	• A single cycle clock cycle is not used because the clock cycle must have the same length for every instruction ◦ The clock cycle is determined by the longest possible path in the processor (the load instruction) ◦ Even though we have a CPI of 1, it is far too long ◦ A single cycle clock cycle violates the key design principle of making the common case fast because it is futile as we must assume the worst-case delay for all instructions
Pipelining	• Pipelining is an implementation technique in which multiple instructions are overlapped in execution, much like an assembly line • Pipelining doesn't decrease the latency of the instruction execution ◦ It is faster because it improves the instruction throughput by increasing the number of simultaneously executing instructions and the rate at which instructions are started and completed ▪ Important because programs execute billions of instructions • The MIPS instructions classically take 5 steps: ◦ Fetch instructions from memory ◦ Read registers while decoding the instruction. TH regular format of MIPS instructions allows reading and decoding to occur simultaneously ◦ Execute the operation or calculate an address ◦ Access an operand in data memory ◦ Write the result into a register • There is a slight start-up and wind-down factor in the beginning and the end of the workload in the pipeline
Hazards	• There are situations in pipelining where the next instruction cannot execute in the following clock cycle. These events are called **hazards** and there are three different types.
Structural Hazard	• A structural hazard occurs when a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute

- This is the reason there are separate instruction and data memories (because we cannot access two different memory addresses in the same memory element in the same clock cycle)
    - Also for temporal and spatial locality discussed in Ch. 5

Data Hazards

- Data hazards occur when the pipelining must be stalled because one step must wait for another to complete.
    - Data is needed to execute the instruction that is not yet available
- For example, if you have an *add* instruction that writes to a register that is an operand for a *sub* instruction that is executed next, we have an issue
    - The *add* instruction won't write to the register until the 5th stage meaning we have to waste 3 clock cycles
- However, the solution comes in the form of **forwarding**, or bypassing
    - This is a method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from the programmer-visible registers or memory
        - In the case of the *add*, the value we seek is ready right after the ALU executes the instruction
        - As soon as this is done, we can forward this by using additional hardware as an input for the subtract instruction
- However, forwarding cannot prevent all pipeline stalls
    - If instead of the *add* instruction, the first instruction was a load, then the desired data would be available only after the fourth stage of load (after we retrieve it from memory)
        - This is too late for the *sub* because we need it as an input of the third stage of the *sub*
- Because of **load-use data hazard**, which is a specific form of data hazard in which the data being loaded by a load instruction has not yet become valuable when it is needed by another instruction, we have to implement a **pipeline stall**.

Control Hazards

- A control hazard, also known as a branch hazard, is when the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed
    - Note that we must begin fetching the instruction following a branch on the very next clock cycle, but we cannot possibly know what the next PC should be because we have just received the branch
- Assume that we can put in enough extra hardware so that we can test registers, calculate the branch address, and update the PC during the second stage of the pipeline
    - There is still one stall that has to be made

- We could implement branch prediction, which assumes a given outcome for the branch and proceeds from that assumption rather than wait for the actual outcome
    - You have to start over if the assumption was wrong
- The approach MIPS uses is **delayed branch**
    - The delayed branch always executes the next sequential instruction, with the branch taking place after that one instruction delay
        - Makes the address relative to PC + 8
    - MIPS software will place an instruction immediately after the delayed branch instruction that is not affected by the branch
        - A taken branch changes the address of the instruction that follows this safe instruction

Implementation of Pipelining

- We separate the datapath into five pieces:
    - IF : Instruction Fetch
    - ID : Instruction decode and register file read
    - EX : Execution or address calculation
    - MEM : Data memory access
    - WB: Write Back
- This separation roughly corresponds to how data flows from left to right, with the exception of:
    - The write-back stage, which places the result back into the register file in the middle of the datapath
    - The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage
- The data flowing from right to left only affects later instructions in the pipeline

- Schematically drawing the pipelining of instructions makes it seem as if each instruction needs its own datapath, but in fact a single datapath can be shared
    - We add registers to retain values between stages and subsequent clock cycles
- The PC can be thought of as a pipeline register
    - Feeds the IF stage of the pipeline
- To pass something from an early pipe stage to a later pipe stage, the information must be placed in a pipeline register
    - For example, in a store we need to pass one of the register contents read in the ID stage to the MEM stage
        - First place it in the ID/EX register and then placed the EX/MEM register
    - For beq, it is relative to the PC so we must pass this into the IF/ID register
    - For the load instruction, in the write back, the register that is changed is currently supplied by the instruction IF/ID pipeline register

- Unfortunately, this isn't the same instruction as the load was for (3 clock cycles later) so we must also carry the register number to be written from the ID/EX through EX/MEM to the MEM/WB pipeline register for use in the WB stage
    - This is also an example of how the right-to-left data flow can cause problems

Pipelining Control

- We must also add pipeline control to the pipelined datapath
    - The control signals from the single cycle datapath carry over, and we re-use them for the ALU control logic, branch logic, destination-register-number multiplexor, and control lines
- We assume that the PC, and each of the pipeline registers, are written during each clock cycle and that no control signals are needed for them
- To specify control for the pipeline, we need only set the control values during each pipeline stage
    - Each control line is associated with a component active in only a single pipeline stage, so we can split them up according to the pipeline stage:
        - Instruction fetch:
            - Nothing special. Control signals to read instruction memory and to write the PC are always asserted.
        - Instruction decode/register file read:
            - Same as IF. The same thing happens every clock cycle, so there are no optional control lines to set
        - Execution/address calculation:
            - Signals to be set are RegDst, ALUOp, and ALUSrc
                - RegDst:
                    - selects the Result register
                - ALUOp:
                    - selects the ALU operation
                - ALUSrc:
                    - selects either Read data 2 or a sign-extended immediate for the ALU
        - Memory access:
            - Control lines to be set are Branch, MemRead, and MemWrite
                - Set by the branch, equal, load, and store instructions respectively
                - Recall that PCSrc selects the next sequential address unless control asserts Branch and the ALU result was 0
        - Write Back:
            - Control lines to be set are MemtoReg and RegWrite
                - MemtoReg:

- Decides between sending the ALU result or the memory value to the register file
- RegWrite:
  - Writes the chosen value

- The simplest way to set the control lines properly in each stage for each instruction is to extend the pipeline registers to include the control information
- Control lines start with EX stage, so we can create the control information during instruction decode

Data Hazards Protection Implementation

- Data Hazards:
- Instructions which have registers as operands that follow an instruction that writes into that register are affected by the fact that the data is not ready for them until during the Write-Back stage of the original instruction
  - There is no hazard during the write-back stage because we assume that register access is fast enough that we are able to write to registers in the first half of the clock cycle and read in the second half
    - Therefore, we are able to use the same register file (eliminating structural hazards) and the register value that is read will be up-to-date
- However, as you know, we can use data forwarding to bypass clock cycles and still get the correct input into the ALU
- There are two pairs of hazard conditions:
  - Either the *rs* or the *rt* registers of the ID/EX pipeline stage match up with the destination register stored in the EX/MEM stage
  - Either the *rs* or the *rt* registers of the ID/EX pipeline stage match up with the destination register stored in the MEM/WB stage
- If this is the case, and the RegWite signal is active for the previous instruction (this only holds for instructions that writes to registers), then we set the forward control respectively for either the *rs* or the *rt* register or both based on whether it comes from the EX/MEM stage or the MEM/WB stage.
  - The *rd* data comes from both registers, into a forwarding unit that accepts the *rt* and *rd* registers of the ID/EX register, and the outputs are the selectors for the multiplexers for the ALU inputs.

- We know that when an instruction tries to read a register following a load instruction that writes to that same register, we need to stall the pipeline by 1 cycle because the data is still being read from memory in the clock cycle while the ALU is performing the operation of the following combination.
- We need to set our hazard control. It takes four inputs:
  - ID/EX.MemRead
  - ID/EX.RegisterRt
  - IF/ID.RegisterRs
  - IF/ID.REgisterRT

- We essentially test if the instruction in the previous cycle is a load by seeing if it's control signal MemRead is asserted, and if it's Rt register, which is the destination register for a load, is the same as either the Rs register or Rt register of the next instruction
    - If this is so, then we prevent the two instructions from making progress by preventing the PC register and the IF/ID pipeline register from changing
        - The back half of the pipeline, starting from the execute, must execute something but we want it to do nothing - essentially, execute a **nop**
        - What we do is set all the control signals to be 0 if the hazard control signal is asserted
            - In this case, the back half of the pipeline won't be doing anything of substance because it won't be changing a register or writing to memory

Control Hazards Protection Implementation

- Our last hazard is a control hazard.
- Note that our decision to branch doesn't occur until the MEM pipeline stage.
    - We improve upon this by moving the branch decision to the ID/ Regfile stage
        - This requires computing the branch target address earlier
            - This isn't an issue because we have the PC and the offset and simply ned to move the brach adder from the EX stage to the ID stage
        - This also requires evaluating the branch decision
            - We need to replace the ALU's role in the ID pipeline stage
            - We can XOR the bits of the two registers and then OR the results for our equality test
        - There are two issues with moving the branch calculation and test to the Decode stage of the pipeline:
            - We need to replicate the forwarding logic which was previously taken care of by the ALU
            - We may need stalls for certain instructions that precede the branch
- If we want to, we could predict the branch and flush the instruction if it's determined to be wrong by zeros the instruction field of the If/ID pipeline register
- However, the MIPS architecture redefines a branch to be a delayed branch where the instruction following the branch is always executed
    - The compilers/assemblers attempt to place an instruct that always executes after the branch in the branch delay slot
        - If we can't find an instruction to put in there, we can put an NOP