Eric, Alex. Bryson. Practice questions.

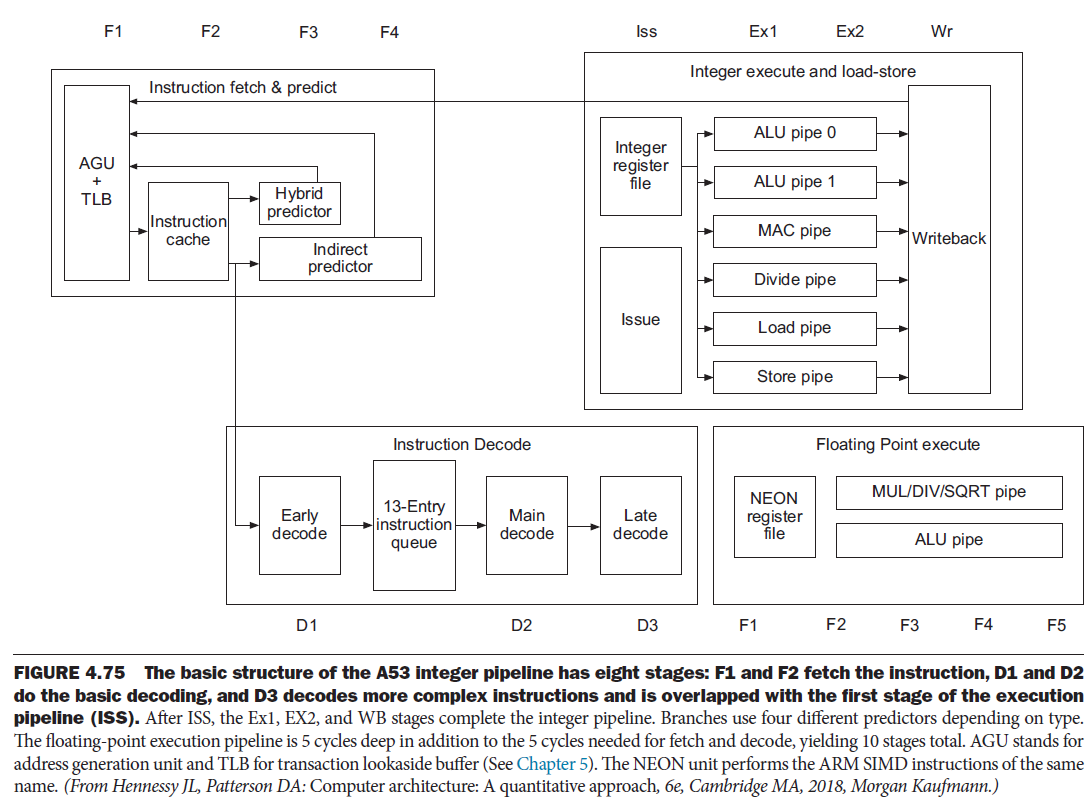
RISC-V Interpeter: <https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/#>

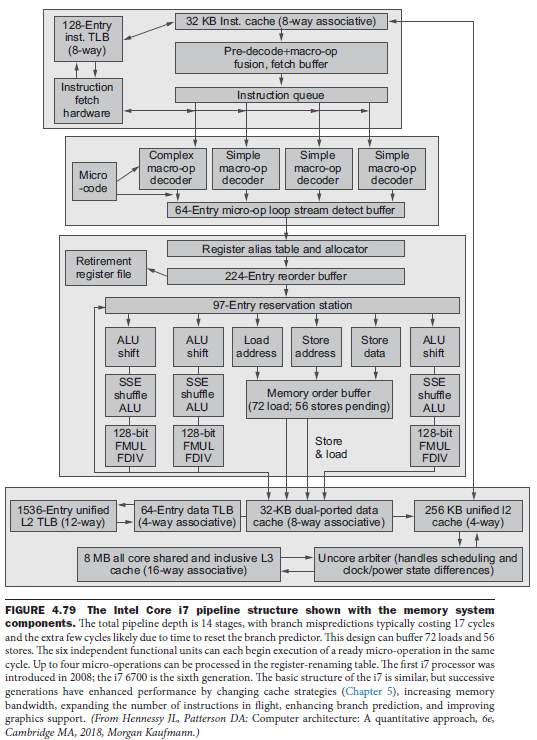
Ch 3

* In general, what does it mean for an arithmetic operation to *overflow*?
  + The bits representing the number in question can't represent the result of an operation with the bits it has.
  + How can overflow be detected in the addition of unsigned numbers?
    - If a bit is carried past the assigned number of bits.
  + How can overflow be detected in the addition of signed numbers?
    - Overflow happens when you add two numbers of the same signs and the most significant bit changes. This would represent a positive number plus a positive number becoming a negative number - overflow.
  + How can overflow be detected in the subtraction of signed numbers?
    - Same concept. You first flip the bits of the one being ‘subtracted’ and add one to it. Then add the two together and apply the same step above.
* What happens in a C program when integer overflow occurs?
  + C and Java both ignore integer overflow when it happens
* In general, what sequence of simpler operations is used to perform integer multiplication?
  + Adders and shifters.
  + Can you step through a multiplication example (with unsigned operands) using the hardware in Figure 3.3?
    - I don’t know, can you? Review this.
  + Can you step through a multiplication example (with unsigned operands) using the refined version of the hardware in Figure 3.5?
    - See previous.
  + What general techniques can be used to produce faster multiplication hardware?
    - Faster multiplications are possible by essentially providing one 32- bit adder for each bit of the multiplier: one input is the multiplicand ANDed with a multiplier bit, and the other is the output of a prior adder.
    - Shifting and adding can also be used to simulate multiplication. (HW 5.14)
    - HW 5 (9-14)
  + What multiply instructions are included in the RISC-V instruction set, and what does each do?
    - RISC-V has four instructions: multiply (mul), multiply high (mulh), multiply high unsigned (mulhu), and multiply high signed-unsigned (mulhsu). To get the integer 32-bit product, the programmer uses mul. To get the upper 32 bits of the 64-bit product, the programmer uses (mulh) if both operands are signed, (mulhu) if both operands are unsigned, or (mulhsu) if one operand is signed and the other is unsigned
* In general, what sequence of simpler operations is used to perform integer division?
  + Also shifting and adding.
  + Can you step through a division example (with unsigned operands) using the hardware in Figure 3.8?
    - Review problems
  + Can you step through a division example (with unsigned operands) using the refined version of the hardware in Figure 3.11?
    - Review problems
  + Why is it more difficult to speed up a hardware divider than a hardware multiplier?
    - A multiplier can perform many additions at once, in a tree format for example. A divider must know the sign of the result of a subtraction before it moves on to the next one.
  + What divide instructions are included in the RISC-V instruction set, and what does each do?
    - Div x5, x6, x7 #Divide
    - Divu x5, x6, x7 #Divide Unsigned
    - Rem x5, x6, x7 #Remainder
    - Remu x5, x6, x7 #Remainder Unsigned
* What support do computers typically provide for representing numbers in a binary form of scientific notation?
  + Float and Float operations
  + What are the conventions for representing *normalized* floating-point numbers?
    - A 1 is assumed to the left of the decimal, all other digits are to the right
  + How are the conventions differ for *denormalized* floating-point numbers? (See the summary in the class slides.)
    - A 0 is assumed to the left of the decimal, all other digits are to the right
  + Given the bit representation of a floating-point number, can you determine the decimal value it represents. (Okay to work with easy examples: 1, 256.5, -15/16, etc.)
    - Practice
  + Given a decimal value, can you determine its bit representation as a floating-point number?
    - Practice
* What steps are required to add two floating-point numbers?
  + 1. Align Decimal points - *Shift number with smaller exponent*
  + 2. Add Significands
  + 3. Normalize Result and check for over/underflow
  + 4. Round & renormalize if necessary
  + What steps are required to multiply two floating-point numbers?
    - 1. Align Decimal points - *Shift number with smaller exponent*
    - 2. Multiply Significands
    - 3. Normalize Result and check for over/underflow
    - 4. If there is overflow or underflow, throw exception
    - 5. Round -> if the rounding takes it out of normalization, go back to step 3
    - 6. Set sign to positive if sign of the operands are same, negative if different
* What floating-point instructions are included in the RISC-V instruction set?
  + Fadd.s (Single add), Fadd.d (Double add)
  + .s and .d for all:
  + Fsub, fmul, fdiv, fsqrt, feq (Float numbers equal)
  + flt (float less than), flte (float less than or equal)
  + Flw, (float load word), fsw (float store word)
  + What do the floating-point comparison instructions do?
    - Sets an integer register to 1 if true, and sets it to 0 if false
  + What registers do the RISC-V floating-point instructions operate on?
    - Floating point registers (f0, f1, …, f31) (f0 is not 0, it is its own register)
* What steps does floating-point hardware take to ensure the accuracy of numerical results?
  + Holds an extra two bits: called Guard and Round, in order to round the least significant bit
  + After taking steps to perform calculation, it rounds based on these two bits
* How can subword parallelism be used to increase the performance of numerical computations?
  + Basically, it allows for calculations of up to 128 bits at a time (sixteen 8-bit operands, eight 16-bit operands, four 32-bit operands, two 64-bit operands)
* Can a shift instruction always replace an integer multiply by a power of 2?
  + No, if a shift results in a change in sign, multiplication failed.
  + Can a shift instruction always replace an integer divide by a power of 2?
    - No. first, the problem of changing a sign can be a problem. Also, negative numbers do not always divide correctly. (See page 238)
* Why are floating-point addition and multiplication not associative?
  + Rounding causes issues. Since all floating points are approximations, errors occur with the associative property. Also, adding two large numbers of opposite sign and one small number doesn’t work. (See page 238)
  + What challenges does this present?
    - This means we have to consider the order of addition and multiplication

Ch 4 part B

* What are real-world analogies for *pipelining*?
  + Laundry, Assembly lines, subway ordering.
  + What is the benefit of pipelining?
    - It reduces total time by working on multiple parts at the same time.
  + Does pipelining reduce the time to execute one instruction, from start to finish (also called the *latency*)?
    - No. It reduces the total time to do all the tasks (Throughput). It doesn’t reduce individual task time (Latency).
  + How does pipelining improve performance?
    - It allows many instructions to be processed at once. In our processor, with 5 states, we can work on 5 instructions at the same time.
* Under what conditions is the speedup from pipelining equal to the number of pipeline stages?
  + Ideal speed up = (time between instructions not pipelined) / (Number of stages)
  + What factors prevent pipelines from achieving the ideal speedup?
    - This doesn’t work exactly because we have unequal stages. Real speed up for 5 stages is closer to x4 Also Hazards
* What characteristics of the RISC-V instruction set suggest that it was designed for pipelined execution?
  + The 5 stages that can each be done in one clock cycle are designed for pipelining. All instructions are the same size and easy to fetch in one clock cycle.
* What sort of dangers or risks arise in the processor because of *pipeline hazards*?
  + What are *structural* hazards, and how can they be eliminated?
    - When a second instruction needs a hardware resource still being used by the first instruction.
  + What are *data* hazards, and how do they arise?
    - Second instruction needs result value produced by the first. In a pipelined system where instructions are executed at the same time, the first instruction won’t write to the register in time for the second one to use it.
  + What are *control* hazards, and what kind of instructions are in the pipeline when they occur?
    - Instruction after branch is determined by the branch result. Therefore, an instruction could start to reassign a register or memory value wrongly if we guessed how the branch behaved wrong
  + It is feasible to rely on the compiler to generate code without any hazards?
    - Going to go out on a limb here and say no…
    - I’m going to say: maybe feasible, but very inefficient. Hardware does a good job at forwarding to avoid hazards. A compiler can catch the rest.
  + What does it mean to *stall* the pipeline?
    - Insert NOPs in the pipeline while we wait for something to occur
  + Can all hazards be dealt with correctly by stalling?
    - Stalling is like adding NOPs (but done by the hardware) to give buffers between instructions using the same register to allow for time for registers to be written to. Stalling can fix all hazards but it is inefficient.
  + What is the key idea behind *forwarding*?
    - The value is forwarded, as soon as it is computed, to the next instruction without needing to wait for the write-back stage. This can eliminate a lot of stalling.
  + Does forwarding eliminate all stalls on data hazards?
    - No. It gets almost all of them. There is still one hazard: The load-Use Data Hazard. This happens when we load a value from memory into a register and then try to use that value in the very next instruction. The data isn’t available until the MEM stage, not the EX stage like most instructions. In this case, we’d still need a stall. We still forward in this case so we only need one stall instead of two.
  + What role can *prediction* play in addressing control hazards?
    - We guess if the branch will be taken or not, and adjust the prediction based on the result. This could mean that we make less mistakes with branches, but it could mean we make more, depending on how branch prediction is implemented.
* Can you provide correct answers to the Check Yourself near the end of Section 4.5?
  + “True, false, or maybe: The control signal PCWriteCond can be replaced by PCSource.”
  + I don’t know if this is the question being referenced. 4.5 is weird in this book. The back of the book has all the answers to the check yourself questions.. Except this one?
* What are the five stages in the pipelined datapath in our text?
  + IF, ID, EX, MEM, WB.
  + What actions are performed in each stage?
    - IF: Instruction Fetch
      * Get the instruction and update the PC
    - ID: Instruction Decode
      * determine Signals for the datapath for the instruction: ALUCtrl, PCSrc, Branch, etc.
      * Get the read and write registers, and the read data.
      * Perform immediate generation
    - EX: Execute
      * Use the ALU to perform proper arithmetic
      * compute the branch target
    - MEM: Memory
      * Read/Write requested memory using ALU result
      * Update the PC if there is a branch
    - WB: Write Back
      * Decide what to write back to register: memory result or ALU result.
      * Write back to the register in the register file
  + What hardware resources of the datapath are associated with each stage?
    - IF: Program Counter
    - ID: Register File, Immediate Generation
    - EX: ALU
    - MEM: Memory read/write
    - WB: Register File
  + Can components of the datapath be used by multiple stages in the same cycle?
    - The ID and WriteBack can both use the register file at the same time. You need multiple ALUs, etc if other components need to be used in more than one stage
  + What hardware resources must be added to the datapath in order to pipeline the execution?
    - Mainly pipeline registers to hold/pass values between stages
* Are pipeline registers part of the visible architectural state of a processor?
  + **No.** See pg 299. “Unlike the shaded pipeline registers in Figure 4.37, however, the PC is part of the visible architectural state; “
  + What are the naming conventions for the pipeline registers? (Which stages does the EX/MEM register separate, for example?)
    - They are named for the two stages they separate. Between the IF and ID stages lies the IF/ID register. (299)
  + In general terms, what information must be stored in each pipeline register?
    - Anything that would be needed in a future clock cycle. For example, the IF stage will store the PC of an instruction in the IF/ID register in case it is needed to calculate a branch target. The instruction is also passed to the IF/ID register so it can be decoded in the ID stage.
    - Other items stored in these registers: Register numbers to be read, the generated immediate, the output of the ALU, data loaded from memory, and any other values that are needed later in the pipeline.
* In Fig. 4.34 (and many others), what is the significance of highlighting (in blue) half of a register or memory used in a given stage?
  + This partial highlighting is a visual representation of which stage is occurring. For example, in the IF stage, hardware the If stage uses is highlighted along with the first half of the IF/ID register. (301)
* What kind of diagrams are used to depict the step-by-step actions of instruction execution in a pipeline?
  + I believe this and the question below asking about 4.41 is really referring to fig 4.45 on pg 308. This diagram shows instructions layed out in a pipeline where each clock cycle executes one stage for each instruction.
  + Multiple-clock-cycle pipeline diagram
  + Given a sequence of RISC-V instructions and the associated multiple-clock-cycle pipeline diagram (as in Figure 4.41), can you determine when data hazards occur? Can you describe how those hazards might be resolved?
    - Yes. Hazards occur when an instruction tries to use a register value that is written to in a previous instruction but the value hasn’t been written to that register yet. This can be fixed using forwarding, NOPs, or stalling.
* Can you provide correct answers to the Check Yourself on page 288?
  + Pretty sure it means pg 309 in the latest edition.
* Are all the control signals used in the single-cycle datapath also used in the pipelined datapath?
  + Yes, we can use the unchanged control format from before. Now they are simply grouped by stage. The pipeline registers are extended to pass on and contain control information. (311)
  + Why are there no write signals for the PC or the pipeline registers?
    - They are always moving with the clock cycle.
  + What control signals are required in each stage of the pipeline?
    - Control signals are generated in the ID stage. EX information is used in the very next stage. MEM information needs to be kept until the MEM stage. The WB information has to be passed through each pipeline register so it can be used in the WB stage.
  + Can control signals be generated before the cycle when they are needed?
    - Yes, they are all generated in ID and carried on until they are needed.
* In the RISC-V pipeline, what happens if a register is read and written in the same clock cycle?
  + We designed the register file specifically to allow us to read the signal being written in the same clock cycle. So you read whatever is being written. This is why two NOPs are sufficient to avoid hazards and we don’t need three. The register is written to in the WB stage and read in the ID stage.
* What does the notation "EX/MEM.RegisterRd" refer to?
  + Signal representing the destination. See if the destination of an instruction is the same as a read register of another future instruction (within 2 instructions)
  + What other values can be referred to using similar notation?
    - MEM/WB.RegisterRd ->
  + Using this notation, how are data hazards identified?
    - The first part of the name, to the left of the period, is the name of the pipeline register; the second part is the name of the field in that register.
    - This notation is described well toward the bottom of page 315. Just go there for this one.
* What is the significance of the blue lines in Figure 4.50?
  + Means figure 4.54 in the textbook. These blue lines are showing which instructions need access to the value being written in the write backstage of the first instruction.
  + In that figure, what do the blue lines that go "backward in time" correspond to?
  + These lines make it easy to see that two of the following four instructions will experience hazards.
* How does forwarding allow the pipeline to run at full speed in the presence of data hazards?
  + It passes on the result of ALU calculations to be used by the next instruction well before it would have been written in the write-back stage.
  + What additional hardware does forwarding require?
    - Forwarding control, extra muxes to decide what is forwarded, pipeline registers.
  + In what stage in the RISC-V pipeline is the added forwarding hardware found? Why is it in that stage?
    - The EX stage. Because the ALU forwarding multiplexors are found in that stage. To go any earlier would require significantly more hardware like another ALU and more ports on the register file.
* Can you explain the significance of each condition and action in the pseudo-code on pages 300-301 (Section 4.7) that details the conditions for detecting hazards and the control signals to resolve them?
  + Practice
* From the Elaboration on page 302, what additional forwarding hardware would be needed in the RISC-V pipeline if we are to avoid stalls on code that loads values and then immediate stores them to a different address?
* In Fig. 4.55, why was a 2nd MUX added before the bottom ALU input?
* What is different about the data hazard caused by a load instruction followed immediately by an instruction that reads the loaded register?
  + How is this hazard detected?
  + What actions are taken in the RISC-V pipeline to get the correct result in this case?
  + Can you explain each condition and action in the book's pseudo-code (on p. 303) to deal with this specific hazard?
* What is a *nop*, and how are nops related to pipeline bubbles?
  + A nop is a sudo instruction that just does nothing. It can be used by a user to put manual breaks in assembly code. Pipeline bubbles, or stalls, are when the hardware puts in a break to avoid various hazards.
  + addi x0, x0, 0
  + What is required to insert a nop or bubble into the pipeline?
    - Nop needs to be input manually by the person coding, or automatically by the hardware in the form of stalls.
* Why are the hazard detection unit and the forwarding unit in different stages?
  + We have to be able to detect hazards in the ID stage in order to update the PC, get values from the proper registers, and control the PC depending on the hazards. The forwarding unit needs to be in the EX stage because it decides what forwarding needs to occur: from MEM or from WB. If a hazard is detected in ID, we know what to forward in the EX stage, to prevent an incorrect calculation
* Do you agree that control hazards are easier to understand than data hazards, but more difficult to address effectively?
  + Yes. Control Hazards occur when the branch does not behave as was predicted. All we need to know is whether the branch will be taken, so that is easy to recognize. Data hazards require close analysis of read registers and write registers, but they can be solved with forwarding. Control hazards require hazard detection, and flushing the pipeline.
* Would it be just as simple for the pipelined RISC-V processor to predict that every conditional branch is taken (rather than not taken)?
  + Not really. In the lab, we predict that every branch is not taken, meaning we just ignore branches and continue to update the PC until the MEM stage has a branch instruction that is taken. To assume that they are taken, we would have to know the branch target address and update the PC to that address as soon as the instruction enters the IF stage, which would be much more difficult.
  + Would this require the calculation of a value that is not required when branches are predicted not taken?
    - Yes, as was described above, the branch target address would need to be calculated.
* What actions must be taken to *flush* or *discard* instructions in the pipeline that follow a mispredicted branch?
  + Turn the other instructions into nops and follow the branch to the new PC.
* What are the benefits of executing branch instructions earlier in the pipeline?
  + What is the earliest stage in which branch instructions could execute?
    - The ID stage.
  + What challenges must be addressed if branches execute in ID?
    - It's only sort of possible. We'd need to have a way for the register file to test if it's read values were equal. It makes for lots of problems. It messes up forwarding. We have stalls that are unavoidable.
* What branch predictors exist that would give better performance than simply predicting all branch outcomes as "not taken"?
  + 1 bit, 2 bit, Tournament?
  + How might the compiler indicate a prediction for a given branch instruction?
  + What is required to implement *dynamic* branch prediction?
  + How is a *branch prediction buffer* (or *branch history table*) accessed, and what information does it contain?
    - Accessed by lookup table, or in a cache, and it contains the branch target address, the branch instruction’s address, and prediction
  + Can you think of a code pattern where a 2-bit predictor is better than a 1-bit predictor?
    - First:
      * Addi t1, x0, 3
      * Addi t0, x0, 0
      * Addi t0, t0, 1
      * Blt t0, t1, First // Always assumes going back because it will most of the time. A 1-bit predictor will be wrong twice per cycle
      * Beq x0, x0, First
  + What advantage does a *branch target buffer* offer?
    - Stores the exact address of a branch target, not just the offset -> can better predict.
    - Also stores the address of the branch instruction, to keep track of perhaps multiple branches with the same target
* What are *interrupts* and *exceptions*?
  + An interrupt is anything external that changes/stops the processor from doing what it was doing.
  + Exceptions are all interrupts as well as the following: External interrupts from hardware devices requesting processor handling. Software interrupts are special instructions indicating that the processor gives up control for a time.
  + What does a processor normally do when an exception occurs?
    - Takes the address of the offending interrupted/exception and saves it in SEPC.
    - It communicates to OS which exception/interrupt occurred.
    - Write value assigned to this location at SCAUSE.
    - Jumps to the handler
* **Summary Steps:**
* Update SEPC and SCAUSE
* Flush following instructions
* Start execution of handler
  + How does the pipelined datapath in Fig. 4.63 respond to an exception?
    - Exceptions in pipelines: Bad memory address(MEM), Illegal instruction(ID), Bad memory address(IF), Arithmetic exception(EX).
    - If an add instruction causes an exception in the EX stage the pipeline must:
      * Allow all previous instructions to complete
      * Flush all instructions in the pipe following the add
      * Prevent x1 from being updated by flushing the add instruction
* What is *instruction-level parallelism* (ILP)?
  + It is parallelism in a single execution thread. It’s very different from multicore parallelism. The most common form of ILP is pipelining.
  + What challenges must be addressed in designing a pipelined processor capable of fetching, decoding, and executing multiple instructions per cycle?
    - You need to read two instructions from instruction mem. Need a lot more ports on the register file. A second immediate gen and a second ALU.
  + How does *static* multiple issue differ from *dynamic* multiple issue?
    - Static Multiple issue: Compiler
      * Compiler groups instructions into “issue packets”
      * Compiler schedules instructions to avoid hazards.
    - Dynamic Multiple Issue: Hardware
      * CPU chooses instructions to issue each cycle
      * CPU detects and resolves hazards at runtime
      * Can still benefit from compiler scheduling
      * Works better with existing programs (no recompilation necessary), works better for branches
  + What does it mean to *speculatively execute* instructions, and how does this differ from simple branch prediction?
  + What additional hardware is required if a pipelined processor is to be made capable of executing instructions *out-of-order*?
* What major differences do you see between the pipelined RISC-V processor in the text, the ARM Cortex A53 pipeline in Fig. 4.75, and the Intel Core i7 920 pipeline in Fig. 4.79?





* + What advanced ILP techniques does the Intel Core i7 use to increase performance?
* Does your experience with the class labs confirm the observation by the authors that creating a correctly executing pipeline can be very subtle (hard to get right)?
  + Yes
* What are some examples of complications in *other* instructions sets (not RISC-V) that make a pipelined implementation more difficult?