Introduction

In this lab, you will develop a RISC-V processor implementing a 3-stage pipeline, along with a sub-set of the RV32I instruction set, along with a test suite validating that your design is implemented correctly.

Specific Tasks

1. Download and extract the project skeleton.
2. Run the shell command `./csce611.sh compile && ./csce611.sh program` to make sure your DE2-115 board is working properly. You should see a bouncing "comet" on the 26 LEDs. If not, or if the script fails, ask a TA for help.
    - You can use this same command to re-compile your project and program the board again at any time.
    - If you see a message like `./csce611.sh: line 47: QUARTUS_ROOTDIR: unbound variable`, then you have not set up your shell session to access the Quartus tools properly. Try running `source /usr/local/3rdparty/cad_setup_files/altera.bash`
3. Design your CPU (see below for details).
4. Modify `top.sv` to instantiate your CPU and connect it to the switch inputs and 7-segment decoders, which are connected to the 7-segement LED outputs.
5. Modify `simtop.sv` to implement a self-checking testbench. If you wish, you may implement several separate test benches, testing separate aspects of your design, or you may keep everything in one, as in the previous lab.
6. Run your testbench using `./csce611.sh testbench`, your own script, or using the ModelSim GUI.
7. Write your binary to decimal conversion program.
8. Deploy your design on the DE2-115 development board using `./csce611.sh compile ; ./csce611.sh program`.
9. When you are satisfied with your design, pack it up using `csce611.sh pack`.

Deliverables

- You will turn in **one** file, it should be named `CSCE611_Fall2021_riu_yourusername.7z`. You should replace `yourusername` with your actual username that you use to log into the Linux lab

Copyright 2021 Charles Daniels, Jason Bakos

computers with. The script will automatically generate a correct file name for you – you usually will not need to modify it.

  - You must turn in your code for this project. Your submission **must** be packed by using the command `./csce611.sh pack`.

- Each group only needs to submit once via either partner.

Objective

In this lab, you will implement a partial RISC-V CPU comprised of a RISC-V ALU, register file, instruction memory, instruction decoder, and various glue logic to implement a RISC-V CPU with support for R-, I-, and U-type instructions. We have provided the ALU and register file for you.

This CPU will only implement a portion of the RISC-V instruction set but will not support features such as load/store, jumping, or branching. To demonstrate the behavior of your CPU, you will implement a simple RISC-V assembler program to convert a binary number entered via the switches (`SW`) to decimal for display on the hexadecimal displays (`HEX0` ... `HEX7`).

In addition, you will define and implement a testing methodology that demonstrates the soundness of your design using testbenches or any other tool of your liking to ensure that all functionality is correctly implemented.

Proposed Approach

This is a large lab which will require you to implement multiple inter-related components. This section outlines a proposed plan of work. You may entirely ignore this section if you wish, but we find that many students struggle with "where to start". To that end, we strongly encourage you to follow the plan laid out here. Note that how you design your CPU is completely up to you, so long as it fits the requirements of this lab and uses the given top-level module. However this section also provides a general overview of one way to structure your processor, which you are encouraged to use as a guide.

First, we suggest you implement your **instruction decoding**. We found it was easiest to do this in it's own stand-alone module, as this makes it easier to test, and will simply things in the next lab when we add J-type instructions. This module should take as input a 32-bit instruction, and should output all of the fields of each type of instruction (see *Appendix B*). This module can be entirely combinational SystemVerilog. We recommend not attempting to determine the instruction type within this module,

Copyright 2021 Charles Daniels, Jason Bakos

and un-conditionally pulling all possible fields out of each instruction[1], and later discarding any un-needed fields elsewhere in your design. This module is a good place to look up whether the instruction is R, I, or U type based on it's opcode, which you can accomplish with a simple lookup table and a `instruction_type` output. Keep in mind that in the next lab, you will need to expand this module to decode B and J type instructions (we do not plan to cover S-type instructions). We suggest testing your instruction decoder using representative but non-exhaustive test vectors[2], and a test bench.

With an instruction decoder, ALU (given with the project), and register file (given with the project) in hand, the final module you may wish to create is a **"control unit"**. This is a lookup table (all combinational, like the decoder) which examines the output of the instruction decoder, and produces the values of any necessary control signals for your design as outputs. "Control signals" will vary from design to design, but these will do things like determine the register file write-enable, select the inputs to the ALU, select the ALU operation, and so on. We suggest creating a large `if-else if-else` in an `always_comb` block. Before the `if`, set the default values for your control lines, and within each case handle the specifics of each instruction. You will find that for many instructions, you will only need to set a few control signals. **It will be easier to just add cases for a few instructions (`addi` is a good first) at first, and come back to fill in the rest later, allowing you to implement instructions incrementally**. Exhaustively testing the control unit would be impractical, but as a sanity check you should feed in a few known instructions and assert that the control lines are as they should be. It's OK if your control unit tests aren't very thorough, as long as your end-to-end tests are, since control unit bugs will also break these.

Suggested control unit outputs:

- `alusrc` - determines which signal should be used as the ALU `b` input in the next cycle between `readdata1`, `readdata2`, and `instruction[15:0]` either sign or zero extended as appropriate.
- `regwrite` - register file write enable.
- `regsel` - selects register file `writedata` between the ALU `R` output the GPIO input, and the shifted immediate field from the U-type format.
- `aluop` - the ALU operation.
- `gpio_we` - enable writing to the GPIO register.

[1] That is, your outputs from this module might include `funct7`, `rs2`, `rs1`, `funct3`, `rd`, `opcode`, `imm_I`, ... We suggest treating I and U opcodes as separate outputs. Keep in mind that a field in an instruction is really just a subset of the bits in a 32-bit value. It is completely safe to extract these bits without knowing what the instruction is – as long as you know before you use them.

[2] Exhaustively testing the instruction decoder would require 2^{32} test vectors. **We suggest creating 32 test vectors – one with a 1 in each possible position**, and checking that it shows up in the right place in each field. For example, a 1 in the most-significant bit of the instruction should result in a 1 bit in the MSB of the `funct7` bit, and in the MSB of the `imm[11:0]` field for I-types.

Copyright 2021 Charles Daniels, Jason Bakos

You will also want to create several 32 bit registers to store intermediate values:

- `gpio_out` - stores the 32 bit value to output via GPIO (updated using `csrrw`), the `gpio_we` signal is the write enable for this register.
- `PC` - the program counter, stores the current address in instruction memory, and increments by one every cycle (in later labs, we will add additional logic to handle overriding it's next value to implement jumps and branches).

To tie all these modules together, we suggest creating a `cpu` module, which takes in a clock, reset, and 32-bit input value, and has a single 32-bit output. Here you can instantiate the modules you have created up to this point, declare all your control lines, and wire everything together. In this part of our design, we append either `_F`, `_EX`, or `_WB` to each `logic` signal according to whether it relates to the Fetch, EXecute, or WriteBack stage. You are strongly encouraged to follow this convention, as it will make it easier to keep things straight.

To see your design through to completion, we suggest implementing just the parts of your CPU and control unit needed for one instruction, writing an end-to-end test, getting it to work, then repeating this process for each remaining instruction one at a time. You really want to build up incrementally – it's very difficult to test and debug if you add hundreds of lines of code all at once.

Note that to support using the HEX displays, you should re-use your hex decocoder module from the previous lab.

Design Requirements

R-Type instruction requirements:

- (i) The instructions add, sub, mul, mulh, mulhu, slt, sltu, and, or, xor, sll, srl, sra, and csrrw must be implemented as described in *Appendix D*

I-Type instruction requirements:

- (iii) The instructions addi, andi, ori, xori, slli, srai, and srli must be implemented as described in *Appendix D*

U-Type instruction requirements:

- (iv) The instructions lui must be implemented as described in *Appendix D*

Instruction Memory Requirements:

- (v) Your CPU must implement an instruction memory of 4096 32-bit words, which should be initialized from a file called `instmem.dat`.

Copyright 2021 Charles Daniels, Jason Bakos

  - **HINT**: `$readmemh()` is synthesizeable and causes the memory it targets to be initialized with the contents of the vector file.

Assembler Program Requirements:

- (vi) Your RISC-V assembler program should be stored in a file named `bin2dec.asm`, which should be a MARS-compatible RISC-V program. It must read a value from GPIO, and output a decimal version of the value via GPIO. It must read and output values in the method prescribed in requirement *(ii)*.

  - **HINT**: you may hard-code input values in RARS, and comment out the line where this occurs for the version of the program you turn in.
  - "Decimal representation" in this context means that when the value is shown in hex, it can be read correctly as decimal, for example, `0x1ef` should be converted to `0x495`.

RISC-V Instruction Pipeline:

- (vii) Your CPU design will implement a pipeline with three stages. This is important because the instruction memory perform synchronous reads, and therefore we must wait one clock cycle after updating the PC for the new instruction to be retrieved.

  - (vi.a) The fetch stage (`F`) will retrieve the next value from the instruction memory.
  - (vi.b) The execute stage (`EX`) will decode the instruction and use the ALU to compute the result of the instructions.
  - (vi.c) The writeback stage (`WB`) will perform writes to the register file. This is needed because when we implement the `load` instruction, the value read from data memory will not be available until after the `EX` stage has completed[3].

General requirements:

- (viii) When `KEY0` is pressed, your CPU should reset itself (i.e. this should be your `rst` signal).
- (ix) A specific testing strategy must be **defined** and **implemented** to ensure that the implemented code performs as it should. This might take the form of a SystemVerilog test bench, a ModelSim TCL script, or some other method. Any test scripts or code must execute on the Swearingen Linux lab computers. A `README` file (or similar documentation) describing how to execute the test suite should be provided. The testing code may assumed that a DE2-115 will be connected to the computer it is running on and may be programmed if needed.
- (x) You must connect and utilize the connections in the `cpu` module as described by their comments and this document.

[3] There is no data memory in this lab, this is just background information

Copyright 2021 Charles Daniels, Jason Bakos

- (xi) When the program counter overflows, it should reset to the value of 0, this is the default behavior of integer overflow in Verilog.

Testing approach

For your testing approach, there are many techniques, one that is straightforward to implement is to write one or more test programs that will result in different values in different registers depending on what instructions are working. You may want some output values that are derived from inputs that have data hazards to test your register file write bypassing. A single test case might look something like:

1. Read an assembled RISC-V program via `instmem.dat` and reset the CPU.
2. Execute the CPU for some number of cycles.
3. Check the state of the registers via testbench against known-good ("ground truth") values.
    - You might establish ground truth by observing the behavior of the simulator.

You might wish to write a shell script or a program in your favorite language to swap out different test programs, and interpret any output your test bench produces (e.g., having your testbench `$display()` the contents of your register file) to decide if the program worked correctly or not.

Rubric

- (A) 30 points – demonstration of bin2dec program operating successfully.
- (B) 20 points – bin2dec program deployed onto the DE2-115 board.
- (C) 30 points – demonstration of your testing suite.
- (D) 20 points – explanation of your testing methodology to the TA.

Maximum score: 100 points.

All grades will be assigned by demoing your code to the TAs. You may demo in any lab session, during office hours, or by appointment. Late penalty will be calculated from the time you turn your code in via Moodle, so it's OK if you don't get to demo before the lab is due.

When you demo your code to the TA, please be prepared to:

- Show your bin2dec program running on the DE2-115 board.
- Show your testbench(es), test vectors, test programs, and any other scripts or materials you use to test your design.
- Briefly explain your testing strategy, and justify why you believe it is a complete and adequate evaluation of your design's functionality. Be prepared to answer questions.

Copyright 2021 Charles Daniels, Jason Bakos

Optional: for rubric category (D), if you feel uncomfortable explaining and justifying your test suite in an oral, in-person setting, you may choose to instead write a textual report, not to exceed 3 pages, to be submitted as a PDF via email to the TAs. You will still need to demonstrate the test suite and bin2dec running however.

Additionally, the following may cause you to lose points:

- Your code may be run through Moss, and your report through plagiarism detection software such as TurnItIn. **Plagiarism and other forms of cheating will be reported to the academic honesty department, which may result in a grade penalty.**
- Submitting code which does not reflect what you demonstrated to the TA's during lab, including code that was modified since it was demonstrated. We reserve the right to verify the code you have turned in against what was demoed, and adjust your score appropriately.

Appendix A - Background

For the remainder of this course, we will focus on developing a RISC-V processor. RISC-V is an open source instruction set. Conceptually, it is very similar to MIPS, which you may have worked with in previous courses such as CSCE212. RISC-V was introduced in 2012 as a standardized instruction set architecture (ISA) which academics and private companies can use as the basis for their own designs. RISC-V is seeing increasing adoption in academia, as well as in private sector.

Previous iterations of this course were built around the MIPS instruction set, due to it's simplicity and widespread use in embedded devices. However the popularity of MIPS has waned even as that of RISC-V has waxed. As a result, we believe RISC-V is more likely to be relevant to any future work you may do in this field than MIPS. However, you will find that the fundamentals of CPU construction are very similar irrespective of ISA chosen.

For the purposes of this and future labs, we will give you the portions of the specification that you require. RISC-V is broken into multiple separate, but related standards. RV32I is a 32-bit integer-only instruction set – we will be implementing many but not all of it's instructions, as well as part of the RV32M multiplication extension. However, if you wish to view the full RISC-V specification, you may do so via this link.

Appendix B - RV32I Instruction Encoding

[RISC-V instruction encoding, sourced from page 12 of the specification.]

Copyright 2021 Charles Daniels, Jason Bakos

NOTE: for this lab, only the R, I, and U type encoding are relevant. We will implement support for B and J types later. S-types will not be used during this course.

Appendix C - Register File Specification

- (i) The register file must have two 32 bit read ports and associated 5 bit read addresses ($\lceil \log_2(32) \rceil = 5$).
- (ii) The register file must have one 32 bit write port and associated 5 bit write address.
- (iii) On any rising clock edge, if the write enable signal is asserted, the value of the write data signal must be written to the memory address specified by the write address.
- (iv) Both read data ports should always have the value stored at the memory address defined by the corresponding read address, except as noted below.
  - (iv.a) If the write enable signal is asserted, then a read via either port of the address defined by writedata should instead yield the value of write data. In other words, both read ports must implement write-bypassing.
  - (iv.b) Any read to the address zero will yield a value of zero via the relevant read data port.
- (v) The register file should have the following inputs/outputs
  - clock
  - reset
  - read data 1
  - read address 1
  - read data 2
  - read address 2
  - write address
  - write data
  - write enable

Appendix D - RV32I R/I/U Instruction Reference

Note: in some cases, instructions are grouped with types other than their actual encoding for logical clarity. Such cases are specifically noted.

Note: all immediate values should be sign-extended. The sign bit of all immediate values is in the most-significant bit (bit 31) of the instruction.

Copyright 2021 Charles Daniels, Jason Bakos

R-Type

These instructions operate on two registers, storing the result in a third register.

31-25	24-20	19-15	14-12	11-7	6-0
`funct7`	`rs2`	`rs1`	`funct3`	`rd`	`opcode`

add

`add rd, rs1, rs2`: R[rd] = R[rs1] + R[rs2]

31-25	24-20	19-15	14-12	11-7	6-0
`0000000`	`rs2`	`rs1`	`000`	`rd`	`0110011`

sub

`sub rd, rs1, rs2`: R[rd] = R[rs1] - R[rs2]

31-25	24-20	19-15	14-12	11-7	6-0
`0100000`	`rs2`	`rs1`	`000`	`rd`	`0110011`

and

`and rd, rs1, rs2`: R[rd] = R[rs1] & R[rs2]

31-25	24-20	19-15	14-12	11-7	6-0
`0000000`	`rs2`	`rs1`	`111`	`rd`	`0110011`

or

`or rd, rs1, rs2`: R[rd] = R[rs1] | R[rs2]

Copyright 2021 Charles Daniels, Jason Bakos

31-25	24-20	19-15	14-12	11-7	6-0
0000000	rs2	rs1	110	rd	0110011

xor

`xor rd, rs1, rs2`: R[rd] = R[rs1] ^ R[rs2]

31-25	24-20	19-15	14-12	11-7	6-0
0000000	rs2	rs1	100	rd	0110011

sll

`sll rd, rs1, rs2`: R[rd] = R[rs1] « R[rs2]

31-25	24-20	19-15	14-12	11-7	6-0
0000000	rs2	rs1	001	rd	0110011

sra

`sra rd, rs1, rs2`: R[rd] = R[rs1] »s R[rs2]

31-25	24-20	19-15	14-12	11-7	6-0
0100000	rs2	rs1	101	rd	0110011

srl

`srl rd, rs1, rs2`: R[rd] = R[rs1] »u R[rs2]

Copyright 2021 Charles Daniels, Jason Bakos

31-25	24-20	19-15	14-12	11-7	6-0
0000000	rs2	rs1	101	rd	0110011

slt

`slt rd, rs1, rs2`: R[rd] = R[rs1] <s R[rs2]

31-25	24-20	19-15	14-12	11-7	6-0
0000000	rs2	rs1	010	rd	0110011

sltu

`sltu rd, rs1, rs2`: R[rd] = R[rs1] <u R[rs2]

31-25	24-20	19-15	14-12	11-7	6-0
0000000	rs2	rs1	011	rd	0110011

mul

`mul rd, rs1, rs2`: R[rd] = R[rs1] * R[rs2]

31-25	24-20	19-15	14-12	11-7	6-0
0000001	rs2	rs1	000	rd	0110011

mulh

`mulh rd, rs1, rs2`: (R[rd] = R[rs1] s*s R[rs2]) » 32

31-25	24-20	19-15	14-12	11-7	6-0
0000001	rs2	rs1	001	rd	0110011

mulhu

`mulhu rd, rs1, rs2`: (R[rd] = R[rs1] u*u R[rs2]) » 32

31-25	24-20	19-15	14-12	11-7	6-0
0000001	rs2	rs1	011	rd	0110011

csrrw

`csrrw rd, csr, rs1`: (t = CSRs[csr]; CSRs[csr] = R[rs1]; R[rd] = t)

31-20	19-15	14-12	11-7	6-0
csr	rs1	001	rd	1110011

I-Type

These instructions are similar but include an immediate value, or a literal constant as the second operand.

31-20	19-15	14-12	11-7	6-0
imm[11:0]	rs1	funct3	rd	opcode

addi

`addi rd, rs1, immediate`: R[rd] = R[rs1] + R[rs2]

Copyright 2021 Charles Daniels, Jason Bakos

31-20	19-15	14-12	11-7	6-0
imm[11:0]	rs1	000	rd	0010011

andi

andi rd, rs1, immediate: R[rd] = R[rs1] & R[rs2]

31-20	19-15	14-12	11-7	6-0
imm[11:0]	rs1	111	rd	0010011

ori

ori rd, rs1, immediate: R[rd] = R[rs1] | R[rs2]

31-20	19-15	14-12	11-7	6-0
imm[11:0]	rs1	110	rd	0010011

xori

xori rd, rs1, immediate: R[rd] = R[rs1] ^ R[rs2]

31-20	19-15	14-12	11-7	6-0
imm[11:0]	rs1	100	rd	0010011

slli

slli rd, rs1, immediate: R[rd] = R[rs1] « shamt

Copyright 2021 Charles Daniels, Jason Bakos

31-25	24-20	19-15	14-12	11-7	6-0
0000000	shamt[4:0]	rs1	001	rd	0010011

srai

srai rd, rs1, immediate: R[rd] = R[rs1] »s shamt

31-25	24-20	19-15	14-12	11-7	6-0
0100000	shamt[4:0]	rs1	101	rd	0010011

srli

srli rd, rs1, immediate: R[rd] = R[rs1] »u shamt

31-25	24-20	19-15	14-12	11-7	6-0
0000000	shamt[4:0]	rs1	101	rd	0010011

U-Type

These instructions are similar to I-type, but include a longer immediate value, and don't accept any source registers.

lui

lui rd, immediate: R[rd] = imm

Note that the lower 12 bits of the immediate value are treated as zero.

31-12	11-7	6-0
imm[31:12]	rd	0110111

Copyright 2021 Charles Daniels, Jason Bakos

Control and Status Register (CSR) Instructions

These instructions read and write to I/O registers. In this course, the IO registers will allow the CPU to interact with the switches, keys, and hex displays of the virtual DE2.

We will define a total of four I/Os, two inputs, and two outputs.

- `io0`, CSR 0xf00, input
- `io1`, CSR 0xf01, input
- `io2`, CSR 0xf02, output
- `io3`, CSR 0xf03, output

`io0` should be connected to the switches, and `io2` should be connected to the HEX displays. You may use `io1` and `io3` for whatever purpose you like, such as for debugging, connecting the LEDs or KEYs, etc.

Note: `io0`, `io1`, `io2`, and `io3` are not standard RISC-V CSRs. A specially modified version of RARS is included with your project skeleton that supports these CSRs. Official versions of MARS will not assemble programs which use these CSRs.

We only need to implement CSRRW for our purposes. Although it ostensibly both writes to and reads from a CSR, the CSRs that we implement are either read-only or write-only, so this instruction can be correctly used to read from or write to any of them. Reading from a write-only I/O such as `io2`, or from an unknown CSR address, should result in a value of `0`. Writing to a read-only I/O such as `io0`, or to an unknown CSR address, should have no effect.

`csrrw`

`csrrw rd, csr, rs1`: t = CSrs[csr]; CSrs[csr] = r[rs1]; r[rd] = t;

31-20	19-15	14-12	11-7	6-0
`imm[11:0]`	`rs1`	`001`	`rd`	`1110011`

Appendix E - Full Instruction Table

For your convenience, the entire table of RV32I and M instructions is given below. Note that includes some instructions we will not use in this course.

mnemonic	spec	funct7	funct3	opcode	encoding
LUI	RV32I			0110111	U
AUIPC	RV32I			0010111	U
JAL	RV32I			1101111	J
JALR	RV32I		000	1100111	I
BEQ	RV32I		000	1100011	B
BNE	RV32I		001	1100011	B
BLT	RV32I		100	1100011	B
BGE	RV32I		101	1100011	B
BLTU	RV32I		110	1100011	B
BGEU	RV32I		111	1100011	B
LB	RV32I		000	0000011	I
LH	RV32I		001	0000011	I
LW	RV32I		010	0000011	I
LBU	RV32I		100	0000011	I
LHU	RV32I		101	0000011	I
SB	RV32I		000	0100011	S
SH	RV32I		001	0100011	S
SW	RV32I		010	0100011	S
ADDI	RV32I		000	0010011	I
SLTI	RV32I		010	0010011	I
SLTIU	RV32I		011	0010011	I
XORI	RV32I		100	0010011	I
ORI	RV32I		110	0010011	I
ANDI	RV32I		111	0010011	I
SLLI	RV32I	0000000	001	0010011	R
SRLI	RV32I	0000000	101	0010011	R
SRAI	RV32I	0100000	101	0010011	R

Copyright 2021 Charles Daniels, Jason Bakos

mnemonic	spec	funct7	funct3	opcode	encoding
ADD	RV32I	0000000	000	0110011	R
SUB	RV32I	0100000	000	0110011	R
SLL	RV32I	0000000	001	0110011	R
SLT	RV32I	0000000	010	0110011	R
SLTU	RV32I	0000000	011	0110011	R
XOR	RV32I	0000000	100	0110011	R
SRL	RV32I	0000000	101	0110011	R
SRA	RV32I	0100000	101	0110011	R
OR	RV32I	0000000	110	0110011	R
AND	RV32I	0000000	111	0110011	R
FENCE	RV32I		000	0001111	I
FENCE.I	RV32I		001	0001111	I
ECALL	RV32I		000	1110011	I
EBREAK	RV32I		000	1110011	I
CSRRW	RV32I		001	1110011	I
CSRRS	RV32I		010	1110011	I
CSRRC	RV32I		011	1110011	I
CSRRWI	RV32I		101	1110011	I
CSRRSI	RV32I		110	1110011	I
CSRRCI	RV32I		111	1110011	I
MUL	RV32M	0000001	000	0110011	R
MULH	RV32M	0000001	001	0110011	R
MULHSU	RV32M	0000001	010	0110011	R
MULHU	RV32M	0000001	011	0110011	R
DIV	RV32M	0000001	100	0110011	R
DIVU	RV32M	0000001	101	0110011	R
REM	RV32M	0000001	110	0110011	R

Copyright 2021 Charles Daniels, Jason Bakos

mnemonic	spec	funct7	funct3	opcode	encoding
REMU	RV32M	0000001	111	0110011	R

Appendix F – ALU Operation Table

This table is also given in the lecture slides, but is reproduced here for convenience.

Note that the ALU B input is used as the shift amount (shamt) for shift instructions, since it would otherwise be unused.

4-bit opcode	function
0000	A and B
0001	A or B
0010	A xor B
0011	A + B
0100	A - B
0101	A * B (low, signed)
0110	A * B (high, signed)
0111	A * B (high, unsigned)
1000	A « shamt
1001	A » shamt
1010	A »> shamt
1100	A < B (signed)
1101	A < B (unsigned)
1110	A < B (unsigned)
1111	A < B (unsigned)

Copyright 2021 Charles Daniels, Jason Bakos