Machine Problem 1: Single-Cycle Processor

2nd sem 2021-22

1 Introduction

This semester's CoE 113 will be working on the RISC-V processor. In order to build a good baseline processor model, we will be implementing a single-cycle version of the RISC-V processor in this machine problem. This model will be useful in verifying functionality of more complex pipelined implementations in the following machine problems for the class.

2 Memory Model

Memory models for program and data memory will be provided for consistency. Although RISC-V uses byte addresses, address pins of the program and data memory will use word and doubleword addresses, respectively. To keep addressing simple, both memories will be using 32-bit byte addresses in little-endian.

For example, if a program memory access is made for (byte) address 0xABCD1234, the lowest 2 bits of the address are ignored while the other address bits are used by the memory. Therefore, an access to word address 0x2AF3448D is made.

Similarly, if a data memory access is made for (byte) address 0x1234ABCD, the lowest 3 bits of the address are ignored. An access to doubleword address 0x2469579 is made.

Truncating the lowest bits of the byte address makes sure that the address forwarded to memory is always word or doubleword aligned for program and data memory, respectively. Thus, a data memory access to (byte) address 0x1234ABCD should access the same doubleword as an access to (byte) address 0x1234ABC8.

Figure 1 shows the interface of the memory model as data memory to the processor. The program memory interface is similar, except that the write ports are not present and the address forwarded by the processor is the program counter.

Reads to both memory models are done by simply supplying a read address to the address (addr) port of the memory. The corresponding read data is combinationally output through the read data port (rdata).

Writes are clock and write-enable controlled. To perform a write to data memory, the write address is supplied to the address (addr) port and the data to be written to the write data port (wdata). A corresponding write enable is sent by asserting (set to HIGH) the wr_en signal. Upon the next clock rising edge, the write is committed.

Some of the instructions in the Machine Problem allows for writes to smaller data sizes, such as words or halfwords. In order to support this functionality, the data memory model will have an additional input named wmask. Each bit of the wmask signal corresponds to a particular byte of the doubleword being written. For example, the doubleword associated with (byte) address 0x1234ABC8 contains 8-bytes, with specific byte addresses of 0x1234ABC8 to 0x1234ABCF. Bit 0 of wmask (wmask[0]) is associated to byte 0x1234ABC8, bit 1 to byte 0x1234ABC9, and so on.

Whenever a write is performed, wmask bits with values of 0 will ignore the write to its corresponding byte. For example, wdata for the write is set to 0xFFFFFFFF_FFFFFFFF and the doubleword to be written to has

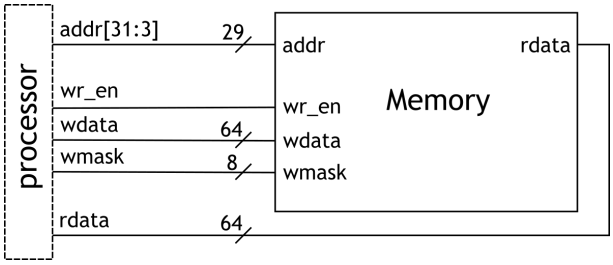
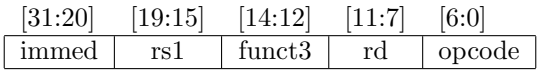
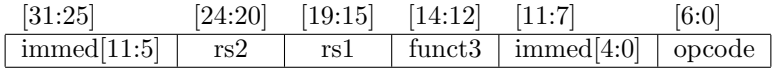


Figure 1: Memory interface



immed:	Address offset (signed 2's complement)
rs1:	Base address register
funct3:	Function code. Equal to 0b011 for ld
rd:	Destination register
opcode:	Opcode. Equal to 0b0000011 for ld

Figure 2: ld instruction endoding (I-type format)



immed:	Address offset (signed 2's complement)
rs1:	Base address register
funct3:	Function code. Equal to 0b011 for sd
rs2:	Source register
opcode:	Opcode. Equal to 0b0100011 for sd

Figure 3: sd instruction encoding (S-type format)

a previous value of 0x00000000_00000000. If wmask has a value of 0x5A, the value of the doubleword after the write becomes 0x00FF00FF_FF00FF00.

If the value of wmask is 0x00, any writes made are ignored. Conversely, a write made with wmask equal to 0xFF writes to all bytes of the doubleword.

The verilog files for the program memory and data memory will be uploaded in UVLE.

3 Instruction Set

RISC-V uses a word length of 64 bits (doubleword) for data and instruction length of 32 bits (word). The following section describes the subset of the RISC-V instruction set that must be supported by your processor.

RISC-V also uses 32 general-purpose, doubleword registers organized in a register file. Each register is addressed by a 5-bit address. When referring to registers in assembly, we will use the notation x[num], where [num] is the decimal equivalent of the register's 5-bit address. For example, register x10 has an address of 0b01010. All registers in the register file are writable, except for x0 which is hardwired to have a constant value of 0.

Unless otherwise specified, all integers are assumed to be encoded in 2's complement.

3.1 Load and Store

We will use the *ld* and *sd* instructions as our base load/store instructions. These instructions transfer doubleword-sized data between the processor and data memory.

The *ld* instruction uses the I-type format for encoding. The fields for this format is shown in figure 2. The *ld* instruction fetches a doubleword stored in the byte address given by the sum of the contents of *rs1* and the sign-extended value of *immed* and saves it to the register specified by *rd*. Since the effective address computed is a byte address, it is up to the programmer to ensure that the effective address is doubleword-aligned.

Figure 3 shows the encoding of the *sd* instruction, which uses the S-type format. The *sd* fetches a doubleword stored in register *rs2* and stores it to a memory location with an effective address equal to the sum of the contents of base register *rs1* and sign-extended *immed*. Take note that the *immed* field is encoded in non-contiguous chunks of the instruction. As with the *ld* instruction, it is up to the programmer to ensure that the effective address is doubleword-aligned.

3.2 Arithmetic

In this machine problem, we will only be using the following register-register instructions: *add*, *sub*, *and*, *or*, *xor*, *slt*. We will also be implementing only one register-immediate instruction: *addi*.

[31:25]	[24:20]	[19:15]	[14:12]	[11:7]	[6:0]
funct7	rs2	rs1	funct3	rd	opcode

rs1: Source register
rs2: Source register
rd: Destination register
funct3: Function code
funct7 Additional function code
opcode: Opcode

Figure 4: R-type format encoding

Instruction	opcode	funct3	funct7
add	0b0110011	0b000	0b0000000
sub	0b0110011	0b000	0b0100000
and	0b0110011	0b111	0b0000000
or	0b0110011	0b110	0b0000000
xor	0b0110011	0b100	0b0000000
slt	0b0110011	0b010	0b0000000

Figure 5: R-type opcode and funct values

3.2.1 Register-register

All register-register instructions use the R-type format, as shown in figure 4. All such instructions take two source operands specified by *rs1* and *rs2*, and store the result to the destination specified by *rd*.

The *add* instruction performs signed addition of the operands. The *sub* instruction subtracts *rs2* from *rs1*, treating both operands as signed. The *and*, *or*, and *xor* perform bitwise and, or, and xor operations, respectively. The *slt* instruction sets the destination to 1 if operand *rs1* is less than operand *rs2* (treating both as signed integers).

Figure 5 shows specific values for the opcode and funct fields for each register-register instruction in our machine problem.

3.2.2 Register-immediate

Register-immediate instructions use the I-type format, same as the *ld* instruction (figure 2). Field *rs1* specifies the register source operand. The 12-bit *immed* field is sign-extended to produce the second source operand. The result is then saved to register *rs*

The *addi* instruction takes the sum of the register and immediate operands. The *opcode* field is set to 0b0010011 and the *funct3* field to 0b000 for the *addi* instruction.

3.3 Conditional Branch

Conditional branches use the SB-type format, as shown in figure 6. Two source registers *rs1* and *rs2* are compared to determine if the branch is taken. The target address is computed by adding the *imm* field to the current Program Counter (PC) value. Take note that the *imm* field's least significant bit (*imm[0]* is not encoded in the instruction, but implied to be 0.

In this machine problem we will be implementing two conditional branches: *beq*, and *bne*. The *beq* instruction takes the branch when both operands are equal, while the *bne* instruction takes the branch when both operands are not equal. Values for *opcode* and *funct3* are shown in figure 7.

3.4 Unconditional Jump

Two unconditional jump instructions will be supported in this machine problem: *jal* and *jalr*.

The *jal* instruction jumps to the effective address computed from the sum of the PC and the sign-extended value of the immediate field. It also saves the address of the next instruction (PC + 4) to the destination register *rd*. Figure 8 shows the encoding of the *jal* that uses the J-type format. Take note that the least significant bit of the immediate field (imm[0]) is not encoded in the instruction. The value of the *opcode* field for *jal* is 0b1101111.

The *jalr* instruction jumps to the effective address computed from the sum of the contents of the source register *rs1* and the sign-extended immediate field. Similar to the *jal* instruction, it also saves the address of

[31]	[30:25]	[24:20]	[19:15]	[14:12]	[11:8]	[7]	[6:0]
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode

rs1: Source register
rs2: Source register
imm: Immediate field. Address offset
funct3: Function code
opcode: Opcode

Figure 6: B-type format encoding

Instruction	opcode	funct3
beq	0b1100011	0b000
bne	0b1100011	0b001

Figure 7: Branch opcode and funct values

the next instruction to the destination register *rd*. The *jalr* instruction uses the I-type format for encoding shown in figure 2, with *opcode* equal to 0b1100111 and *funct3* equal to 0b000.

4 Top-level module and interface

The processor to be implemented must execute all instructions in a single clock cycle. All writes to the register and data memory must be committed in the next rising edge of the clock after the program counter is set by the current instruction. Figure 9 shows the required external ports of the processor and their corresponding widths. Take note that although the memory model discussed previously only accepts doubleword addresses, you are still required to output the lowest 3 bits of the program counter (pc) and the data address (addr). In addition to the memory interfaces, a global clock and reset signal is present. The reset signal (nrst) is low-asserted, and resets all general-purpose registers and the pc to 0 when asserted.

Synthesizability in Xilinx Vivado is a requirement for evaluation. Make sure that your code is synthesizable and run appropriate tests on the synthesized model of your system. You may use the Artix board configuration in CoE 111 as the target device for configuration purposes.

5 Grading

Grading will be based on the functionality of instructions. Criteria for grading this machine problem will be as follows:

- Load and Store - 40 points
- Arithmetic: Register-register - 20 points
- Arithmetic: Register-immediate - 10 points
- Conditional branches - 15 points
- Unconditional jumps - 15 points

Full points for a category will be given only if the synthesized model works for that category. Half points will be given instead if the RTL model functions correctly. Testing for all non load/store categories are dependent on the functionality of ld and sd, so make sure that your ld and sd instructions work perfectly.

[31]	[30:21]	[20]	[19:12]	[11:7]	[6:0]
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode

imm Immediate field. Address offset
rd Destination register for address of next instruction
opcode: Opcode

Figure 8: J-type format encoding

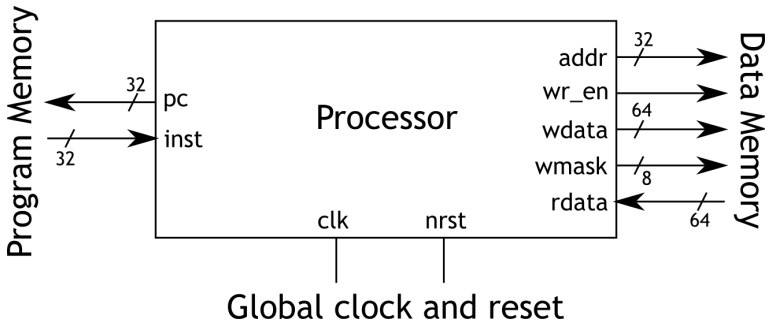


Figure 9: Processor Interface

Deadline for submission will be on April 21, 2022, through an UVLE submission bin. Upload all source files in an archived file. Do not upload the entire Vivado project directory. Allowed archive formats are: zip, tar, tar.gz. Failure to comply to these rules will automatically merit a 25 point deduction. Late submissions will be allowed, but with a 10 point deduction for each week late.