Module Descriptions:

Instruction Fetch Stage:

The IF stage is responsible for two essential functions for each instruction: first in stores and updates the program counter with either PC+4 or a pc to be branched or jumped to from a later stage. Second it sends the current PC, byte_enable, and a read signal to the instruction cache to grab the instruction. Both the PC and the instruction are necessary in the Instruction Decode Phase.

Input Signals:

PC_IF: The output of the register holding the current pc as remembered by the IF stage

ADDRESS_MEM: the address to branch to as output by the EXE stage and stored in the Mem Control Regs

PC_MUX_SEL: chooses between pc+4 and Address Mem

CACHE_READ_DATA_IF: the data response from the instruction cache: the current instruction

CACHE_RESP_IF: A single bit signal from the instruction cache indicating that

CACHE_READ_DATA_IF is valid given MEM_BYTE_ENABLE, READ_IF, and PC_IF

Output Signals:

PC_IF_NEXT: The PC of the next instruction

READ_IF: A single bit signal to the instruction cache requesting the data at the address indicated by the PC_IF_REG

MEM_BYTE_EN: 4 bits which are always high and determine which of the 4 bytes of the address being read from are ewS

CACHE_DATA_IF: The word of data being read from the Instruction Cache which is the current instruction

Instruction Saved Registers:

This Modules saves the current value of PC_IF and the Instruction. PC_IF is needed by the IF Stage to calculate PC+4 and by the ID Stage to pass forward to the EXE Stage for some alu

ops. The Instruction is needed by the ID stage to determine almost every control singal and input which gets sent forward to EXE.

Regs:

PC_IF: next value determined by PC_IF_NEXT

CACHE_DATA_IF: next value determined by CACHE_DATA_IF from IF

Instruction Decode Stage:

The Instruction Decode Stage determines the control signals for all future stages based on the pc and the instruction.

Note that the PC, RS1_OUT, RS2_OUT, and other signals are present to the EXE stage but this stage selects among those signals which will be needed as ALU and COMP inputs. They are captured by the below output signals precisely when they are needed in later stages.

Also note that we have only accounted for one possible hazard with the WRITE_REG_MEM_WB, ALU_OUT_MEM_WB, and RD_MEM_WB signals. This design doc is not required to and does not yet attempt to catch and correct all hazards.

Inputs:

PC_IF: the value of PC, sometimes used by EXE for the alu

PC_JUMP_IF: the value of PC determined by a jump in the instruction in the EXE stage.

BR_EN: Indicates thata jump has occurred and we should take PC_JUMP_IF not PC_IF+4 as the next instruction and that we should squash the current instruction.

CACHE_DATA_IF: the Instruction itself to determine the signals for the next stage

WRITE_REG_MEM_WB: A signal piped in from the later MEM_WB stage as an **example** of hazard handling. Indicates that a register was modified by the instruction in MEM_WB (3 instructions in the past)

ALU_OUT_MEM_WB: A signal piped in from MEM_WB stage as an example of hazard handling. This is the data that a register was set to if any register was modified at all. This will replace rs1_out or rs2_out if rs1 or rs2 are the register which was modified.

RD_MEM_WB: A signal piped in from MEM_WB as an example of hazard handling. This is the index of the register which was modfied if any was. This is compared with rs1_id and rs2_id to see if either has been modified by the instruction 3 instructions in the past but not saved to the regfile.

Outputs:

WRITE_REG_ID: TForwarded eventually to MEM_WB to let that module know that this instruction needs to write to RD.

WRITE_ID: Forwarded eventually to the MEM stage to indicate that it needs to write to memory

READ_ID: Forwarded eventually to the MEM stage to indicate that it needs to read to memory.

CMP_OP: Determines whether the comparator should perform unsigned or signed, equality, greater than or less than comparison between its inputs.

ALU_OP: Determines whether the comparator should perform addition, XOR, OR, AND or several other operations on ALU_IN_1 and ALU_IN_2.

RD_ID: index of the RD register which will be used later to write to the regfile and will be piped back to future instructions' ID stages in case there RS1 or RS2 have been modified by the current instruction.

CMP_IN_1: The first of two inputs that will go to the comparator. This happens to be exactly RS1_OUT.

CMP_IN_2: The second of two inputs that will go to the comparator. Selected by an internal mux from RS2_OUT and PC.

ALU_INPUT_1: The first of two inputs that go to the alu. Selected by an internal mux from several inputs

ALU_INPUT_2: The second of two inputs that go to the alu. Selected by an internal mux from several inputs.

RESET_STAGE: A signal that is set high if this instruction is a jump or a successful branch instruction in order to squash the next two instructions so they don't corrupt the run state.

Instruction Decode Control Registers:

Stores the output from the ID stage for exe to use. All outputs of ID are registers and outputs to EXE with the exception of RESET_STAGE.

Execution Stage:

The execution stage contains the alu, and the comparator and is used to calculate arithmetic results to be written to the regfile, values for interfacing with memory, or the pc_mux value for the IF stage in case of a branch or jump.

NOTE: We foresee but do not here account for a hazard where the instruction 1 in the past (currently in the mem stage) modifies rs1_out or rs2_out for this instruction. We have to handle that in the exe phase because the new value might not be determined until after memory reads and exe has to have access to it. MEM_WB->EXE and MEM->ID are both two cycles so MEM->EXE is the only pipe that fixes that hazard.

NOTE: THE ALU is not exactly identical to the state machine implementation. This one also calculates comparisons and the cmp only ever decides branches. ALU_OP is not equal to FUNCT3.

Inputs:

WRITE_REG_ID: A signal which indicates that this instruction will eventually modify rd. This will be important for MEM_WB stage to have.

WRITE_ID: A signal which indicates for MEM that this instruction means to write to memory. Forward.

BR_EN: A signal indicating that jump has occurred and this instruction is invalid and should be squashed.

READ_ID: A signal which indicates for MEM that this instruction intends to read from memory. Forward.

CMP_OP_ID: Determines which type of comparison CMP is performing on CMP_IN1 and CMP_IN2.

ALU_OP_ID: Determines what kind of arithmetic operation the alu is performing on alu_input_1 and alu_input_2.

RD_ID: The index of the destination register so MEM_WB knows where to result data and future instructions know which register was modified.

CMP_IN1: The first of two input to the comparator. Will be overridable in case of a hazard.

CMP_IN2: The second of two inputs to the ALU. Will be overridable in case of a hazard.

ALU_INPUT_1: The first of two inputs to the ALU. Will be overridable in case of a hazard.

ALU_INPUT_2: The second of two inputs to the comparator. Will be overridable in case of a hazard.

Outputs:

ADDRESS_MEM: The address which mem will read from or write to if it is to interface with the dcache.

BR_EN_MEM: The result of the comparator operation. This either is the new value of the reg, rd or it indicates that a branch has occurred.

ALU_RESULT_DATA_MEM: The output of the alu to be written to the regfile.

READ_MEM: Indicates that mem should read from dcache.

WRITE_MEM: Indicates that mem should write to dcache.

RD_MEM: the index of the register to be written to

WRITE_REG_MEM: A signal indicating that RD should be modified

MEM_BYTE_EN_MEM: The byte mask for the memory unit to use.

Memory Control Registers:

All of the output signals from exe are stored here for piping and for mem stage.

Memory Access Stage:

Inputs:

ADDRESS_MEM: The address to read or write to in dcache.

BR_EN_MEM: A signal indicating to the IF stage and the ID_stage that a branch has occured

ARITH_OUT: The result of either the cmp or the alu if this instruction writes an arithmetic result to a register

READ_MEM: A signal indicating that mem should read from dcache

WRITE_MEM: A signal indicating that mem should write to dcache

RD_MEM: Indexes which register is to be written to and helps future instructions decide if there was a hazard

WRITE_REG_MEM: A signal that indicates that this instruction should write to the regfile.

MEM_BYTE_EN_MEM: The byte mask for memory accesses

From Cache:

MEM_RDATA: the data from dcache on a read

CACH_RESP_D: a bit signal from dcache indicating that MEM_RDATA is valid or that a write was successful

Outputs:

RD_MEM_WB: The index of the reg for the MEM_WB stage to write to and for future instructions to detect hazards

WRITE_REG_MEM_WB: A signal indicating that this instruction will write to the regfile

To Cache:

WRITE_TO_CACHE_D: Indicates to the cache that a write is occurring.

READ_FROM_CACHE_D: Indicates to the cache that a read is occuring

MEM_WDATA: The data to be written to the cache

MEM_BYTE_EN: The byte mask for a memory access.

ADDRESS_D: The address to be accessed

MEM WB REGISTERS:

All the outputs from the mem stage that do not go to cache are saved in registers.

MEM_WB_STAGE:

This stage either does nothing or writes the result of a read or arithmetic operation to RD.

Inputs:

RD_MEM_WB:The register index we may be writing to and used for detecting hazards in previous stages.

WRITE_REG_MEM_WB: A bit signal indicating that we should write back to the regfile and indicating that there may be a hazard for previous stages

RD_DATA_IN: The data to be written to RD.

Design Decisions, possible optimizations, and foreseen bugs/hazards:

- Hazard: Consecutive instructions rs1 or rs2 == rd
- Hazard: One instruction in between instructions rs1 or rs2 == rd
- Bug: lock step wait mechanism on data miss or instruction miss
- Optimization: Constant memory access prefetching (predictable address misses
- Optimization: ID Signal Truncation
- Optimization: combine MEM_WB and MEM
- Optimization: use pc_jump immediately not next cycle in IF
- Optimization: LRU cache optimization
- Optimization: CMP and ALU combination
- Optimization: Dont Write to RD if another instruction intends to (because pipelining covers it and there is no need to toggle the reg if another instruction will overwrite it right away
- Design Decision: Combine Arithmetic CMP functionality into the alu.