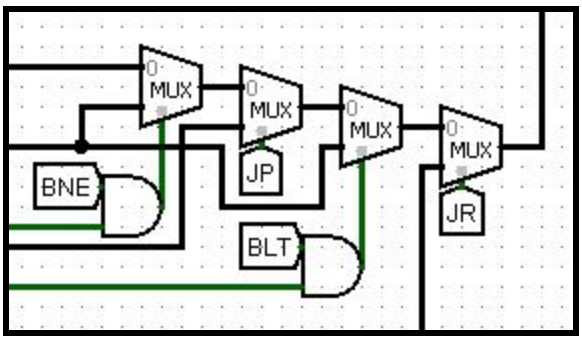
Overview

This processor operates under a five-stage pipeline, meaning the overall computational flow is broken into five chunks, separated by pipeline latches (registers). The latches are as follows:

Program Counter (32-Bit)

The input to this register is the current 32-bit value of the program counter. This can be either a normal incremented Program Counter (PC + 1), a branched program counter (PC + 1 + N), or a jumped program counter (PC = T). The selection of these different possible program counters is done with multiplexers that contain potential branch/jump values based on decoded instructions later in the pipeline. This discrimination among program counter values can be seen in the following circuit diagram, where the final multiplexor output is fed into the input of the program counter register, in order to properly fetch the correct instruction from ROM.



The output of the program counter register is fed into the address field of the instruction memory, in order to fetch the correct instruction from the ROM array. The fetched instruction and incremented program counter are then fed into the next latch in the pipeline.

Fetch/Decode (64-Bit) (Top Half - PC, Bottom Half - Instruction)

This stage of the processor involves the deconstruction of the fetched instruction in order to lookup the appropriate registers in the Register File, serving mainly to populate the 5-bit crtl_ReadRegA and ctrl_ReadRegB fields. This is because different instruction types use different parts of the instruction to dictate which register needs to be accessed. For instance, R-type instructions contain both an $rs and an $rt, which means both read register fields must be filled with the [21:17] and [16:12] bits of the instruction, respectively. However, I-type instructions may require a plethora of lookups. Load, Store Word, and add immediate, for example, only need to lookup $rs. Branches, however, require both $rd and $rs to be looked up, since they need to be compared later in the pipeline. Additionally, bex needs to look up the value in $rstatus ($30). This discrimination among the possible registers to lookup is done with a chain of multiplexors, similar to the program counter implementation. The program counter, both results of the Register File Lookup (data_readRegA and data_readRegB), and instruction are then funneled into the next stage of the processor.

Decode/Execute (128-Bit) (Program Counter, Reg A Value, Reg B Value, Instruction)
This stage contains the bulk of the computational logic of the processor. One of the fundamental purposes of this stage is to populate two operands destined for the Arithmetic Logic Unit (ALU). Considering the variety of instruction types, these two operands may take different forms. In an r-type instruction, operands A and B will directly be the values of the Registers looked up in the previous stage. In add immediate, load word, and store word, however, operand A will remain the same, but operand B must be a sign extended integer from the Immediate field of the instruction. The branch instructions have already been accounted for in the previous stage, when $rd and $rs were looked up in the Register File.

This stage also contains much of the program counter anomaly logic, including for branches and jumps. In the case of branches, the ALU has two outputs iLT and iNE that compare the two operands. These outputs are utilized (only if the current instruction is a branch) in order to choose between a normal program counter and a new program counter + N (sign extended immediate value). My processor implements branch recovery, which means the current instructions in the F/D and D/X latches are flushed (replaced with nops) in order to quickly branch to a new instruction in the assembly code.

Furthermore, a significant amount of stall logic is found in this stage, and will be covered in a later section of the report. The result of the ALU, an untouched version of data_ReadRegB (in the case of store word, where the contents of $rd must be written to memory), and the instruction are then passed on to the next stage.

Execute/Memory (96-Bit) (ALU Result, Register B Value, Instruction)
This stage exists almost exclusively for the load word and store word instructions, both of which require the processor's memory (RAM). The data memory module is fed both an address (result of ALU addition) to lookup and potential data to write to memory (in the case of store word). In a load word instruction, the output of the data memory module (given the appropriate address) is the value that will eventually be written to a register in the next stage. If one thinks of memory as an array of elements, then the address fed into the memory block can be thought of as an index in the array. In a store word instruction, the value at this address is replaced with the contents of a register. In a load word instruction, the value at the address is simply looked up and passed along to the next stage. Since all other instructions do NOT require this stage, the untouched ALU Result, the fetched data from memory (load word), and the instruction are passed along to the next stage.

Memory/Writeback (96-Bit) (ALU Result, Data from RAM, Instruction)
This stage deals primarily with the logic for writing a value back into a destination register in the Register File. There are two main outputs that are populated in this stage, ctrl_writeReg and data_writeReg, which specifies to the register file which register will be written with what value.

It is in this stage that the discrimination among $rd, $r31 (for jal), and $rstatus occurs, with each instruction type having a unique register number and value. In a load word instruction for example, the data result from RAM must be written to a register, so a multiplexor chooses between the ALU Result and the Data result. In a jump and link instruction, the 31st register must be written to, so a multiplexor makes sure that ctrl_writeReg is equal to 31 in the case of a jal opcode. Similar logic is implemented for setx and the status register. The memory/writeback stage is crucial in order to store values within our processor's register files, enabling us to write lines of assembly code that can refer back to past instructions.

Hazard Detection/Stalling

My processor utilizes a combination of stalling and WX bypassing in order to mitigate the impact of data hazards. Due to the nature of the five-stage processor that advances with the clock, the processor can only read register values **three** cycles after it has been written. To enforce that assembly code with rapid register retrieval doesn't break, the instructions in each stage are examined, as to stall the processor until the register value is ready to be accessed. This is done with the physical insertion of a null operation (nop) as a potential instruction input to the F/D register, which delays the processor by one cycle, hopefully enough to resolve the data hazard. The stall logic checks the register to be accessed from the F/D stage and compares it to the destination register of the D/X stage. If they are equivalent, or there is too close of a gap between loading and storing a word, then the processor will stall by physically inserting a nop into hardware. The logic is represented by the following equation.

```
Stall = (D/X.IR.OP == LOAD) &&
        ((F/D.IR.RS1 == D/X.IR.RD) ||
        ((F/D.IR.RS2 == D/X.IR.RD) && (F/D.IR.OP != STORE)))
```

Additionally, my processor implements bypassing in order to bypass/update the register file. This is done by reading a value from an intermediate source (such as a future stage), rather than waiting for the value to be available from the register file. My processor implements bypassing for both ALU operands, as well as for the store data input, so that load and store word instruction may be written back-to-back in assembly code. ALU operand A can either be the ALU Result from the next stage, the data value that is being written to the Register file, or the standard input from the Register File. This basically intercepts the required values mid-stage instead of waiting for them to be written to Register File. Similar logic is implemented for ALU operand B, but this time with potential $rt or immediate values.

Improvements to be Made

- At present, my processor is incapable of multiplying and dividing using my modified booth algorithm multiplier and non-restoring divider. Possible paths for completion could include:
    - Create a new co-latch (P/W) as a coprocessor containing both a multiplier and divider. If the current instruction is of opcode mul or div:
    - Stall 16 cycles (using FSM counter from multiplier) until multReady output from modified Booth multiplier is asserted.
    - Stall 32 cycles (using FSM counter from divider) until divReady output is asserted.
    - Pipeline the product/quotient into the final writeback mux to choose the multdiv result instead of the normal writeback result/load word result.
- Although the logic for checking the value in the $rstatus register is implemented, overflow logic to actually update the register value has NOT been implemented at present. Steps to include this are:
    - Because overflow logic requires two registers to be written in the same instruction, include a stall such that if the overflow result of the ALU is asserted, stall, write to $r30. The value for data_writeReg could be determined with a 5-input multiplexor that chooses the appropriate status value (1-5) depending on the opcode of the instruction that produced the overflow.
- There is currently a bug with the store word pipelining logic that requires a hardcoded nop statement (add $0, $0, $0) after every store word instruction within the assembly/instruction memory file. Various testing was conducted looking at the enable signals for the q_dmem, but no solution was found within the given time constraint. Possible fixes could include:
    - Re-evaluating current store word logic
    - Adding a store word instruction as another possible nop case in which a stall is required
- Jump instructions have not been rigorously tested, so there may be significant bugs that have not been found. The jump instructions were added using the same general idea as branches, just without the conditional checks.