Final Project :: ECE 350

Matt Kleiser

Branch Predictor and Fast Branching

Git Repository: https://github.com/mhk29/ECE350-Final-Project

Overview:

This project aimed to make improvements to the MIPS-like-processor project completed earlier in the class, particularly focusing on branching, one of the slowest aspects of the previous design. The two most major pieces of improvement were the execution of branch instructions during the Decode stage rather than the Execute stage as had been previously built, and the inclusion of a branch predictor to anticipate the result of branches.

The fast branching part of this process included the addition of bypassing into the Decode stage from both the Execute and Memory Stages, and the movement of the branching logic to the Decode stage from the Execute stage. The flushing of the program pipeline also was adjusted to not inject a nop into the pipeline registers between the Decode and Execute stages, ensuring that the pipeline could execute branches more quickly. The last major aspect of this was implementing immediate comparison between two 32-bit binary numbers, for both the branch-not-equal and branch-less-than instructions.

The branch prediction part of this process involved both the creation of the branch predictor and the implementation of the branch predictor into the remainder of the processor. The branch predictor itself was built first, implementing an eight-instruction two-bit saturating counter direction predictor. This branch predictor was built to update only on mispredictions and initializes its direction prediction upon detection of a not previously seen branch instruction to either strongly taken or strongly not taken based on the result of the initial direction taken by the branch. Registers are used in conjunction with a series of sets of two D-flip-flops to create a lookup table for the two-bit saturating counter. A 6-bit hash based on the least significant 6-bits of a 32-bit PC# for table lookup.

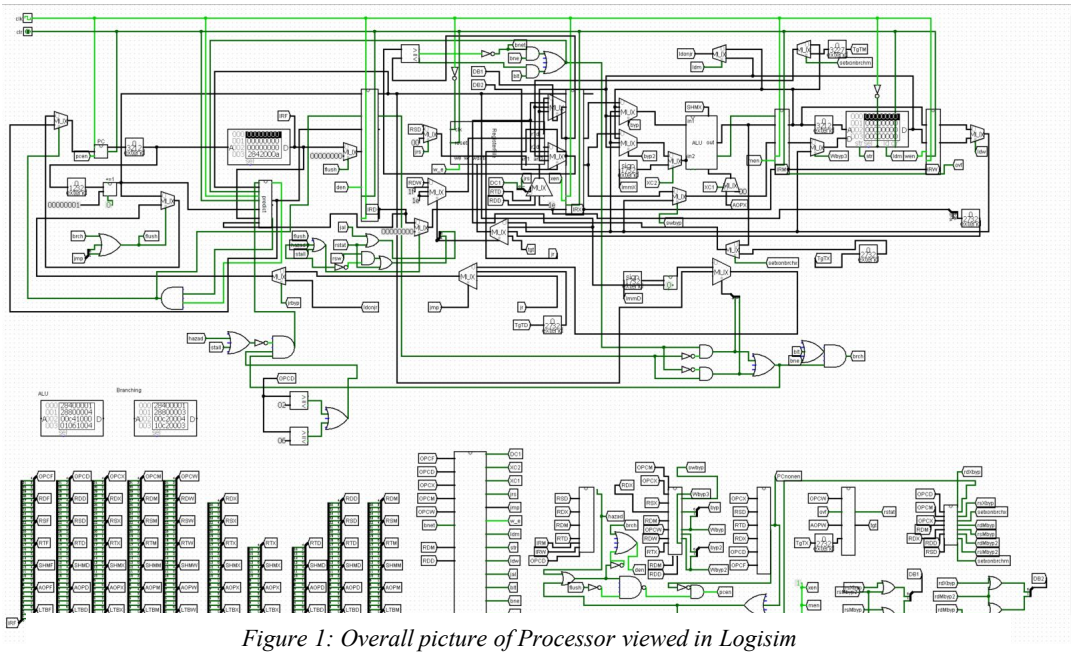


Figure 1: Overall picture of Processor viewed in Logisim

Overall, this process required significant changes to the Decode stage of the processor, the branching and jumping methodology of the

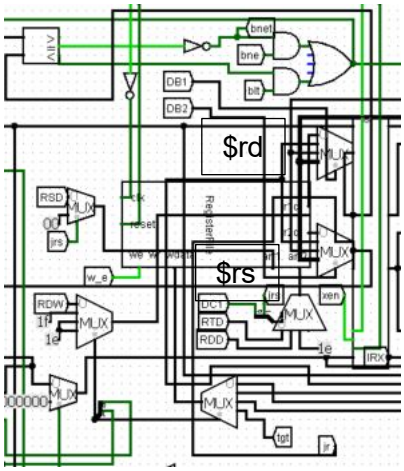
processor, as well as the control logic. Significant changes in the bypass control, including introduction of bypass logic into the decode stage led to major additions to the processor.

Fast Branching:

Fast branching was the first major task undertaken to improve the processor and was implemented prior to the building of the branch predictor. Several rules were made to govern the building of the fast branching before it was built, and these rules were as follows:

Bypassing from Execute to Decode:

- If an ALU operation (i.e. add, addi, sub, etc.) occurs one instruction before a branch or a jump and $rd in Execute stage equals the $rd or $rs in the decode stage, then bypass into the appropriate $rd or $rs MUX in the decode stage
- If a lw occurs one before a branch or jump operation and the $rd in the lw operation equals $rd or $rs in the branch operation, a stall operation must occur to prevent a data hazard
- If a sext occurs one before a branch or jump if $rd or $rs in branch instruction equals $r30=$rstatus, bypass target (bits 26:0) into appropriate $rd or $rs MUX in the decode stage



Bypassing from Memory to Decode

- Bypass into the appropriate $rd or $rs MUX in the decode stage if $rd in the memory stage equals $rd or $rs in the execute stage
- If lw occurs two operations before, bypass $rd in the memory stage into either $rd or $rs of the decode stage if $rd in the memory stage equals either $rd or $rs of the decode stage
- If setx occurs two instructions before a branch or jump operation, execute a bypass similar to and ALU Op bypass from the memory stage into the decode stage

All of the above required the creation of an additional logic unit to hold all of this logic. Some of the logic in the existing bypass control unit was adjusted to take the additional stall into consideration, with certain comparisons of registers being changed, notably if bypassing from a lw result into the Execute stage. Other minor changes were made to make the circuit appear more aesthetically pleasing, including the neatening of wires due to a very wire-dense decode stage resulting from the addition of fast branching. A single comparator was used to determine if branch-not-equal or branch-less-than should activate rather than the outputs of the ALU. If a branch recovery must be performed, the pipeline registers between the Decode and Execute stages are not cleared as they previously were before fast branching. Jump instructions are also executed in the Execute stage with minor changes to target addresses and Register Jump (jr) registers being in the Decode rather than execute stage. After some additional testing described below, it was found that the branches appropriately executed and the pipeline executed properly.

Branch Prediction:

An eight-instruction lookup table-based two-bit saturating counter direction predictor was implemented for branch prediction. The lookup table was built using eight registers containing eight different instructions for branch prediction. When new branch instructions are detected by the branch predictor, the instruction is stored in the lookup table. When the instruction is executed, a corresponding two-bit saturating counter is updated to strongly predict whatever the result of the first instance of the branch is (if taken or not taken after first execution). For all comparisons performed in the lookup table, a six-bit hash consisting of the final six bits of the program counter number corresponding to the instruction. This hash is based on the natural log of the number of bits of the length of the instruction, and was recommended directly by Dr. John A. Board as an appropriate number of bits for a hash for performing comparisons. A 32-bit to 6-bit "extender" was used to take the last six bits of any program counter number.

To detect if an instruction has been seen before, a series of comparators compares each of the hashes from the registers to a hash of the program counter number (PC#) being updated. If none of the program counter numbers in any of the registers correspond to the instruction that needs to be updated, then a counter increment by one and then activates the appropriate register for a new instruction. For example, for the third-detected branch instruction, as the instruction is detected the counter increments by one from two to three, and the register for the third instruction is activated holding that PC#. Additionally, the third two-bit saturating counter will be set to either be equal to 00 (strongly not taken) if the branch is not taken the first time or 11 (strongly taken) if taken the first time.

All branch prediction updates occur one cycle after a prediction is made, as branches are executed in the Decode stage of the processor pipeline, and updates only occur if a misprediction is made (i.e. the outcome of the branch is different from the prediction), and if a hazard or stall is not activated. This means that all branch instructions must be given a prediction before the lookup table can provide a branch prediction result. Therefore, if a branch instruction is not previously seen before, a branch is predicted to be taken in all cases. If a branch misprediction occurs, a pipeline flush of the Decode stage occurs, and the branch predictor is updated according to the standard two-bit saturating counter schema in Figure 2.

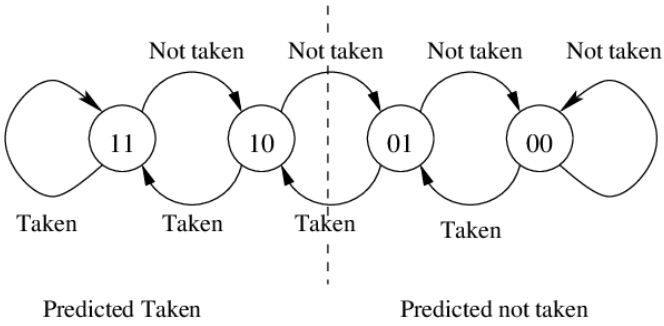


Figure 2 Two-bit saturating counter schema

For performing predictions, instructions on which a branch is performed are branch-less-than (blt) and branch-not-equal (bne). The branch-on-exception (bex) instruction was specifically not included in this branch prediction scheme since the bex instruction is executed using the same way jump instructions are with an additional condition making the use of this branch predictor difficult without more major changes to the processor and potentially adding a great amount of combinational logic.

To check if a branch has already been detected by the lookup table, a series of comparators between the 6-bit hash of the output of the registers and the current PC to see if there is a match. If so, the output of the two bit saturating counter is output. If no match is found, the output is set to have a branch be taken (see Challenges more details).

In order to output the proper target as a result of a branch, an adder is included in the branch predictor to add a number from the last 17-bits of the instruction to the PC# + 1 and output either the original PC# + 1 or the taken target depending on if a branch instruction is predicted to be taken (and if a branch instruction is detected at all in the current instruction). This works fairly consistently and since all branch updates occur one cycle after a misprediction. After somewhat extensive testing, the branch predictor appeared to work as intended.

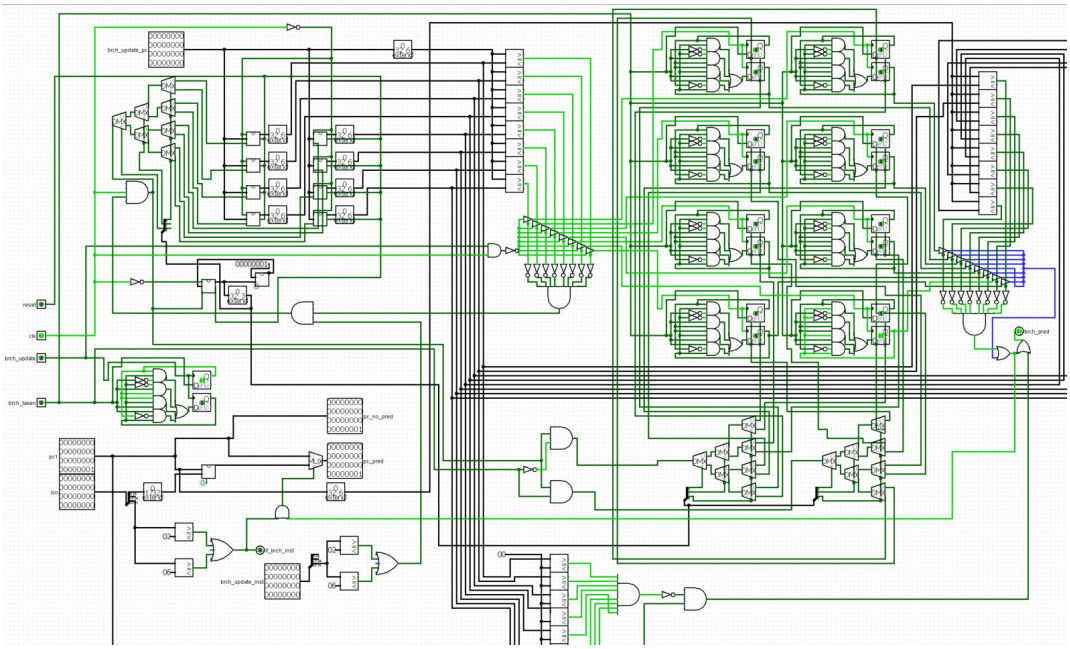


Figure 3: Branch-prediction scheme with registers

Testing and Testing Programs:

Four major test programs were used, one for branching, one for jumping, one for bex and setx, and one for other miscellaneous instruction (entitled: Practice). Each of the test programs were modified slightly for various cases, but the final test cases used are included below.

Several branch prediction and fast branching cases were tested using the branch programs, including having a lw before a branch, a branch followed by an add followed by a branch, and a sw (store word) two instructions before a lw. Several branches were also executed in this program to detect if predictions were occurring properly and if the pipeline was being flushed on mispredictions only.

Both the jump and bex/setx programs led to the expected results with no additional changes being required. The practice program included jump instructions before branches, as well as a series of instructions containing a setx-jal- and blt in immediate succession. This uncovered certain challenges which were corrected.

Challenges:

The practice program revealed three major sets of challenges.

1. The Last Instruction (PC = fff)
2. A jump instruction occurring before a branch instruction
3. Storing more in the pipeline registers between the fetch and decode stage

It was found that the branch predictor had be initialized to 0, and this could lead to problems with the first instruction being numbered PC=000 in the program counter. In order to make the last instruction no longer a problem, all PC#s were stored as the current PC# + 1 in all cases for the branch predictor, with all comparisons being made against the current PC# + 1. However, this pushed this issue to the last instruction (PC=fff), meaning something had to be done. If it was found that this instruction is reached, the instruction is added like any other instruction to the program buffer, with the standard counter incrementing. This requires a special detection scheme to ensure that this is stored properly. The instruction is checked to see if any register currently contains PC=000, and if so, this is noted. If a branch instruction is in this position, the same scheme ensures that the branch is set to be predicted.

If a jump instruction occurs before a branch instruction, the branch instruction occasionally would become lost in the pipeline and not executed. This required the addition of logic to ensure flushes would only occur if a branch instruction was being executed in the Decode stage, rather than if a branch instruction had been predicted. This was ultimately found to be the result of the branch predictor being set to predict a branch and this not being properly cleared in the pipeline, but another workaround was used based on detection of the bne and blt instructions occurring in the Decode stage from the existing control module.

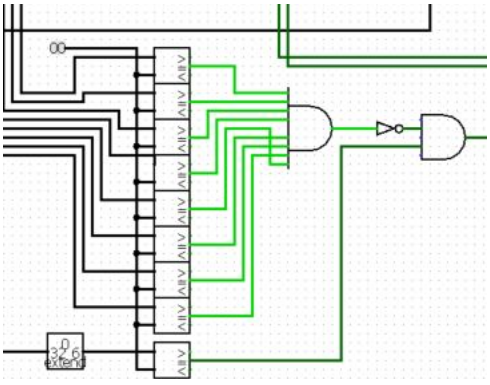


Figure 4: PC=fff exception handling logic

Adding more registers to the FDpipe module was somewhat challenging due to the difficulty of not adding a flush function to the branch prediction inputs, however, adding combinational logic to detect if a branch instruction was occurring in the Decode stage fixed this issue.

Potential Improvements:

The most major improvement that could be made is to perform logic minimization of the current logic. Certain sets of logic include up to six gates of combinational gate delays. This would create a more apparent issue if this processor were programed in Verilog or built in a lab setting, but this is not a major issue in Verilog. Similarly, more was considered for addition to the branch predictor, such as adding a global predictor for additional branch prediction, however, no major additions were made to completion.

Miscellaneous

Brch.s

```
.text

nop                 # 0
addi $r1,$r0,1      # 1
addi $r2,$r0,3
add $r3,$r1,$r0
bne $r3,$r1,3       # 4
add $r4,$r2,$r1
bne $r4,$r1,3       # 6
addi $r8,$r1,1      # non-ex
add  $r9,$r1,$r2    # non-ex
add  $r10,$r6,$r2   # non-ex
sw $r4,5($r1)       # 10
add $r5,$r2,$r3
lw $r5,5($r1)       # 12
blt $r5,$r2,1       # 13
add $r7,$r6,$r2
sub $r0,$r0,$r0             # 15 last op

# non-ex reg: r8,r9,r10
# note that the above commentary is likely inaccurate
```

Jmp.s

```
.text

nop                 #0
addi $r1,$r0,1      #1
addi $r2,$r0,3
add $r3,$r1,$r0
sw $r2, 5($r1)
add $r9,$r1,$r2
jal 9               #4 -- data hazard must be caught
lw $r31, 5($r1)
jr $r31
add $r8,$r1,$r2
add $r4,$r0,$r1             # non-ex
addi $r5,$r0,1      #
j 13                #7
```

```
add $r4,$r0,$r2         # non-ex
addi $r5,$r0,2          # non-ex
jr $r31                 #10
add $r4,$r0,$r3                 # non-ex
addi $r5,$r0,4          # non-ex
sub $r0,$r0,$r0                 #13 lsat op

# check to make sure writeback still works
# with jal instruction
# above commentary may be incorrect logically
```

Etx.s

```
.text

nop                     # 0
addi $r1,$r0,1          # 1
addi $r2,$r0,3
setx 5                  # 3
addi $r5,$r0,1          #non-ex
bex 8                   # 5
setx 0                  # 6
bex 9                   # 7
addi $r5,$r0,2          #non-ex
addi $r4,$r0,1          #force-ex
setx 11                 # 10
sub $r4,$r4,$r4
sw $r4,5($r1)           # 12
lw $r30,5($r1)          # 13
bex 17                  # 14
addi $r5,$r0,4          #non-ex
addi $r5,$r0,8          #non-ex
sub $r0,$r0,$r0                 # 17 last op

# non-ex reg: r5
# force-ex reg: r4
# above commentary may be incorrect logically
```

Prac.s

```
.text

nop                 # 0
addi $r1,$r0,1      # 1
addi $r2,$r0,3
addi $r31,$r0,9
setx 9              # 4
j 6
bne $r30,$r31,4
setx 1              # 7
jal 8
blt $r31,$r30,-10
nop
```