Hazard Unit &
Branch Prediction

Laboratory Project - Pipelined Processor with Hazard Unit and Branch Predictor

Objectives

The aim of this project is to expand upon the design of the 32-bit pipelined processor of the 4th lab. You will extend the previous datapath and the control unit of the pipelined processor **with the hazard unit and the branch predictor**. For this project the instruction set is extended with a few more instructions. The designed processor will be able to execute all instructions in the **extended** instruction set.

Finally, you will embed your design into the FPGA of the DE1-SoC board and demonstrate your design.

This project will be done in groups of 2 students unless you chose to do the project by yourself. You can choose your partner. Each partner is expected to contribute in equal amounts. If the work is divided too unevenly, the student who did more work will be generously graded while the student who did less work will be penalized. The most uneven work division acceptable is 60-40. If you cannot find a partner we will match you with another student (if possible).

1 Project Work

1.1 Reading Assignment

If you feel unfamiliar with pipelined CPU architectures hazard unit and/or branch predictor, please refer to the corresponding **lecture notes of EE446** course.

1.2 Baseline for the Project

In order to implement the branch predictor and the hazard unit you will need a working pipelined processor, which is laboratory 4's work. You can work together with your project partner to help each other (merge your codes, debug your codes, etc.) to complete the 4th lab.

1.3 ISA to be implemented

The processor you will design will not support all ARM instructions as in the previous laboratory works but only a restricted set listed in Table 1. For all instructions, conditional logic of **EQ, NE, and AL** are required. Condition codes defined in ARM standards are shown in Figure 6. You will implement a shifting functionality for the second operand for data processing. Except for MOV, logical shift right and left functionality is required for only data processing instructions, the "sh" value shown in Figure 5 will be 00 for LSL and 01 for LSR.

The new MOV with immediate: The immediate operand rotate field is a 4-bit unsigned integer that specifies a shift operation on the 8-bit immediate value. This value is zero-extended to 32 bits, and then subject to a rotate right by twice the value in the rotate field. This enables many common constants to be generated.

Note: Instructions that move another register to PC such as BX will be implemented but the branch predictor will not handle this, it will be handled by the hazard unit as in the lecture notes. The computer will flush 2 stages if the branch is taken as in lecture notes.

Mnemonic	Name		Operation
ADD	Addition	add Rd,Rn,Rm	Rd← Rn + (Rm *sh* shamt5)
SUB	Subtraction	sub Rd,Rn,Rm	Rd← Rn - (Rm *sh* shamt5)
AND	Bitwise And	and Rd,Rn,Rm	Rd← Rn & (Rm *sh* shamt5)
ORR	Bitwise Or	orr Rd,Rn,Rm	Rd← Rn | (Rm *sh* shamt5)
MOV	Move to Register	mov Rd,Rm	Rd← (Rm *sh* shamt5)
MOV	Move to Register	mov Rd,rot-imm8	Rd← (imm8 *rr* rot<< 1)
STR	Store	str Rd,[Rn,imm12]	Mem[Rn + imm12] ← Rd
LDR	Load	ldr Rd,[Rn,imm12]	Rd ← Mem[Rn + imm12]
CMP	Compare	cmp Rd,Rn,Rm	set the flag if (Rn - Rm =0)
B	Branch	b imm24	PC ← (PC + 8) + (imm24<< 2)
BEQ	Branch if Equal	beq imm24	PC ← (PC + 8) + (imm24<< 2) if flag = 1
BL	Branch with Link	bl imm24	PC ← (PC + 8) + (imm24<< 2), R14 ← PC + 4
BX	Branch and Exchange	bx Rm	PC ← Rm

Table 1: ISA to be implemented

Note: For this lab, you will use the 32-bit ARM ISA format as shown in Figure 5.

1.4 Datapath Changes (25%)

In this part of the project, given your ISA, you are expected to design a full datapath that would support all the instructions included in Figure 5. Using the pipelined processor implementation from the laboratory 4, you will implement a hazard unit and a branch predictor **You can use Verilog HDL to construct your datapath.**

Some changes are expected in your modules, these are:

- Change the register File such that it writes to the registers on the negative edge of the clock.
- For the data memory you will use the SDRAM of the DE1-SoC board rather than your own memory module.
- Extra Muxes to forward data for hazards and branch prediction target addresses.

You are allowed to use the following components to construct the datapath :

- Instruction Memory
- SDRAM as Data Memory
- Register file
- Program Counter register
- ALU
- Immediate Extender
- Multiplexers
- Combinational Shifter
- Interim registers for pipelined operation
- Adders

Give your reasoning in the report for the changes you made to the datapath in the lecture notes. You can change the datapath as much as you want as long as you give proper reasoning. As a rule of thumb, try not to forward data between stages unnecessarily and use the inter-stage registers as much as possible to ensure the critical path is small.

Considering the instruction set provided in Table 1, perform the following design steps:

1. (4%) Describe/explain how data processing instructions are executed and data hazards handled in your datapath as specified in Table 1
2. (3%) Describe/explain how memory instructions are executed and data hazards handled in your datapath as specified in Table 1
3. (3%) Describe/explain how branch instructions are executed and control hazards handled in your datapath as specified in Table 1
4. (15% Credits) Implement the final datapath, which supports all the required instructions in **Schematic Editor of Quartus.**
    - (5% Credits) Capability of executing data processing instructions showed in Table 1
    - (5% Credits) Capability of executing memory instructions showed in Table 1
    - (5% Credits) Capability of branch instructions showed in Table 1

1.5 Hazard Unit (25%)

The hazard unit that will be implemented for this project will handle two hazard types. Most of the design will be consistent with the implementation you have studied in the lectures. As a reminder, the list of the terms for hazard handling is given.

- Flush: Clearing a stage register so that the result of that stage is discarded
- Stall: Holding the value of a stage register so that a bubble can be introduced
- Forward: Sending the calculated value to a previous stage

1.5.1 Data Hazard Handling (15%)

Data hazards happen when an instruction tries to read a register that has not yet been written back by a previous instruction. There can be multiple methods to handle this hazard type even as simple as constant stalling. However, you can see that this implementation method will decrease the efficiency of your design. Hence you are required to implement your hazard unit such that:

- Hazards caused by data operations must be handled by forwarding such that no cycle is wasted.
- Hazards caused by memory operations can use a minimal amount of stalling.

1.5.2 Control Hazard Handling (10%)

Control hazards happen when the decision of what instruction to fetch next has not been made by the time the fetch takes place. In this project, different branch types will use different methods.

- Branch operations that use immediate values (B, BL, and their conditional variants) will use branch predictor (see subsection 1.6 and lecture notes "Ch4_2 BRANCH PREDICTION_2023"). However, the predictor can make wrong guesses in which case flushing the stages are also necessary
- Branch operations that use register values (MOV r15, BX, and their conditional variants) will forward the new PC value to the fetch cycle and flush the wrong stages **when the branch is taken**. This should be implemented with a minimal amount of flushing while considering the critical path. See lecture notes "Ch4 ARM PIPELINED_2023".

1.6 Branch Predictor (25%)

The branch predictor implemented for this lab will consist of 3 parts. Branch target buffer for storing the PCs and branch target addresses of the last branch instructions, global branch history register, and pattern table. The overall structure of the predictor can be seen in Figure 1.

The predictor will have a RESET input which will be used at the beginning to initialize the predictor.

The branch predictor can make wrong predictions, hence the computer will sometimes branch when it should not. In this case, Fetch and Decode stages should be flushed and PC + 4 after the branch instruction should be fetched. Thus, apart from the 3 modules that will be discussed predictor will also forward PC + 4 from the branch instructions till the Execute stage for possible use.

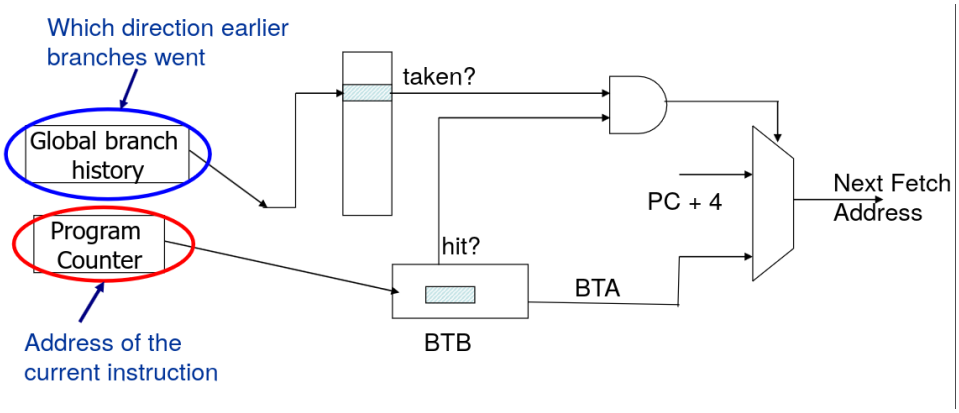

Figure 1: Overall Branch Predictor Layout

To realize this predictor you will be using Verilog HDL, **Schematic editor is not required apart from the extra Mux in the datapath that will select the next fetch address**.

Now, let's discuss each part of the predictor separately.

1.6.1 Branch Target Buffer (10%)

To store the branch target addresses (BTA) for the predictor you will implement a BTB. BTB is a simple structure that stores PC values for the branch instructions and corresponding BTA's. BTB will also have a comparator circuit for the stored PC values. The design will look like Figure 2.

BTB will have 2 outputs, a 1-bit hit flag and BTA. BTB will take the current PC as input and compare it to the saved PCs, any hit will result in the hit flag being raised and the corresponding BTA to be given. BTB you implement will contain 3 entries for easy demonstration.

Apart from looking for matches, this module should have a way to add entries. Adding new entries when the table is full should overwrite the oldest used entry, (similar to LRU in caches) meaning the recently used entries should be protected.

The implementation specifics of this module are up to you but you must add a reset signal such that the table can be initialized.

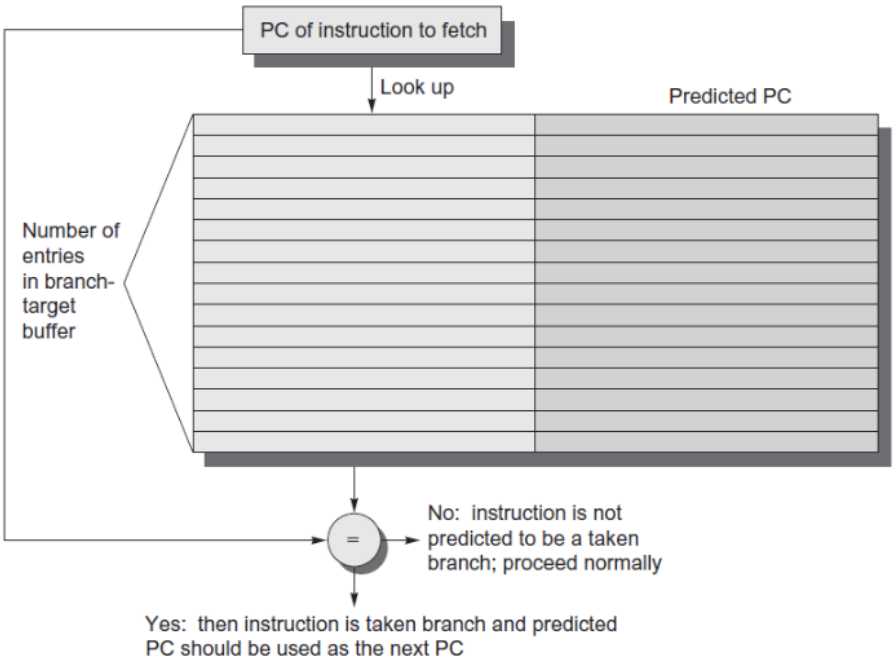

Figure 2: Branch Target Buffer Layout

1.6.2 Global History Register (GHR) (5%)

This part is the simplest, it is just a shift register which shifts in 1 for taken branches and 0 for not taken branches. It can shift from the right or left both will be the same in practice but we will make it shift left. **For easy demonstration the register will be 3-bits wide.**

Reset	Shift	Operation
1	X	DATA $\leftarrow$ 0
0	0	DATA $\leftarrow$ DATA
0	1	DATA $\leftarrow$ {DATA<< 1,In}

Table 2: Operation of the GHR

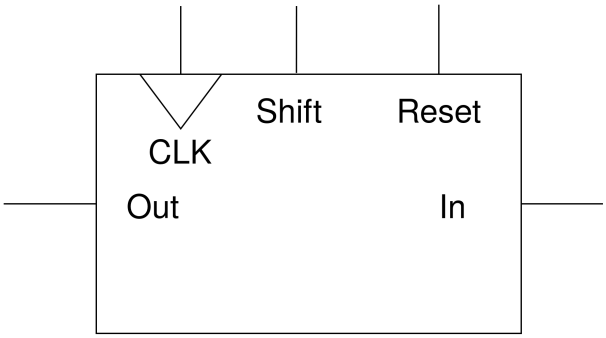

Figure 3: Global History Register

1.6.3 Pattern History Table (PHT) (10%)

Pattern table is where the computer stores if a branch was taken or not when a pattern of previous branches shown in the global register was encountered. Since our GHR is 3-bits wide the **PHT will contain 8 elements each of which is 1-bit wide.** If an element is 1 it means the branch was taken with that history, if it's 0 branch was not taken. So the entries of the PHT are 1-bit last time predictors.

The implementation specifics of this module are up to you but you must add a reset signal such that the table can be initialized.

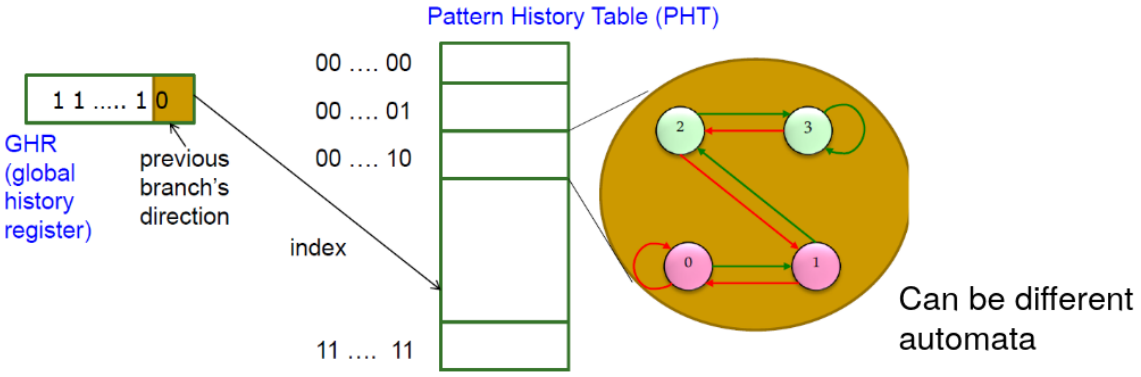

Figure 4: Branch Target Buffer Layout

1.7 Testbench (25% Credits)

Now that you have completed the implementation of the pipelined computer, it is required to verify its operation through some light programming. These programs will be some small test codes that you will write to test the full execution of a code. Be sure to write some code that demonstrates all the possible hazard types and the working of the branch predictor correctly.

After constructing the code, write a cocotb testbench that checks for the correct operation of the computer. A proper testbench is automated, so it should indicate when something fails in the design without needing any manual work. You can use prints/logs in your testbench for debug purposes.

Explain the code in your report, and point out the hazard and prediction points in the code. Show with your testbench that the design handles hazards and predictions correctly.

- (5% Credits) Testing the datapath changes without any hazard or prediction
- (10% Credits) Testing the hazard unit functionality of the computer
- (10% Credits) Testing the branch predictor functionality of the computer

2 Project Demonstration

Load your processor designed in the Preliminary Work Part 1.2 to the DE1-SoC board. In the lab you will be given a code segment for your computer, you will upload the code and run it to demonstrate to your teachers.

A Useful Materials

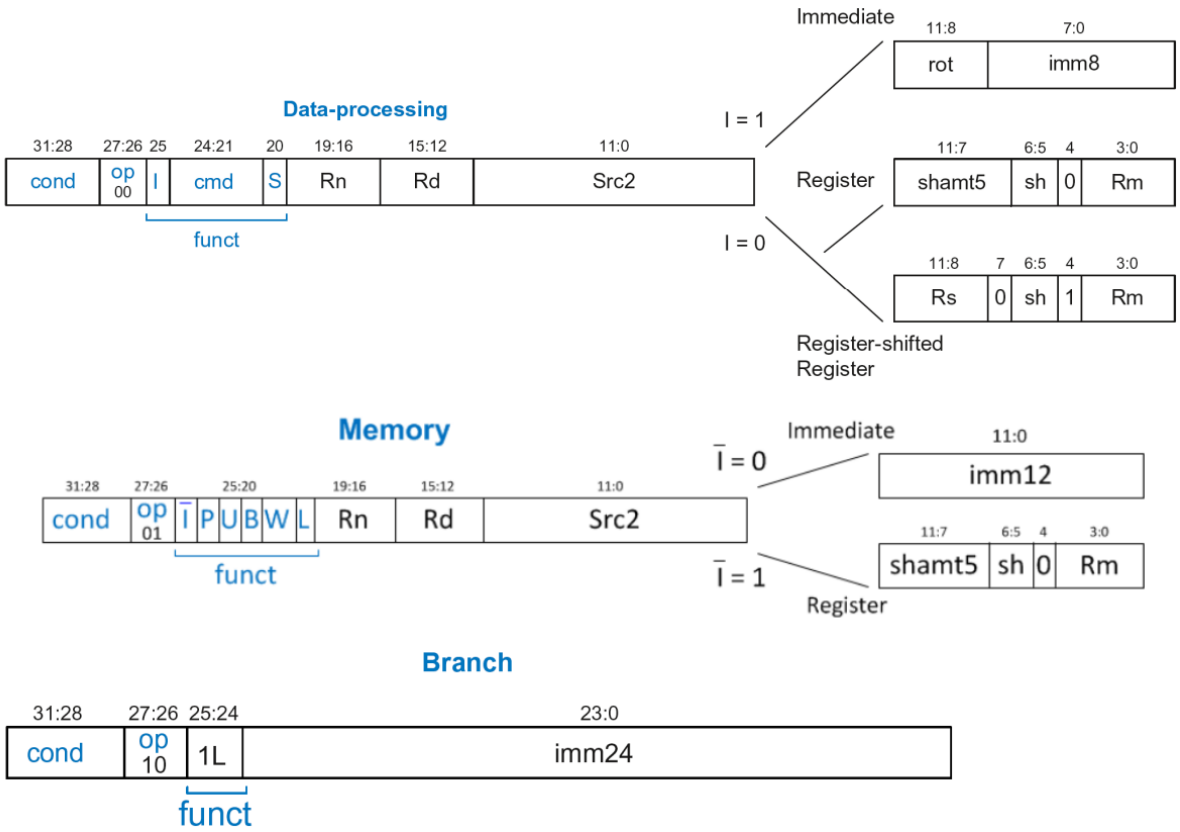

Figure 5: ARM ISA Format

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

Figure 6: ARM Condition Codes

B Parts List

`DE1-SoC Board`