Designing a Custom Pipelined Processor

Tyler D'Angelo

A project presented for the degree of
Master of Science in Computer Science and Engineering

Supervised by Dr. Kris Schindler

Department of Computer Science and Engineering
University at Buffalo
Buffalo, New York, United States
December 2025

Abstract

The goal of this project was to design a complete processor with the ability to write and run full assembly language programs. The first step was to design a custom RISC instruction set with only the necessary and most useful instructions. The next step was to design the assembler. This is meant to be a straightforward mapping of assembly instructions to their binary machine code. The bulk of the project was to design the processor itself. This was done in Verilog and run on the Basys3 [2] FPGA board. As the board has a few peripherals, the ability to utilize buttons and a VGA display was also included in the architecture. The final step was to write a game to play using the processor. This would show that the processor could be reasonably used to write software.

1 Introduction

This paper details the design of the assembly instruction set, the assembler, the processor itself, and programs written for the processor. It also goes into some of the thought behind design decisions.

Section 2 covers the design of the custom RISC instruction set. It details the instruction format and the instructions implemented.

Section 3 covers the implementation of the processor. It goes through the full datapath, stage by stage. At each stage, it also covers the major components: what they do and how they are used with respect to the rest of the architecture.

Section 4 covers the assembler. There are a few additional considerations beyond a direct mapping to binary that are explained. Pseudo-instructions are detailed. In addition, labels are also automatically converted to a binary address offset for branch instructions.

Section 5 covers an example program using all parts of the architecture. *Atari Breakout* [4] was written using the custom instruction set and run on the Basys3 [2] FPGA board.

2 Instruction Set

2.1 Instruction Format

Table 1 illustrates the instruction formats followed for this processor.

As this is a 16-bit processor, there are 16 registers. Each instruction interacting with registers needs 4 bits for each register referenced. The registers are labeled *RS* as the source register, *RT* as the target (or secondary source) register, and *RD* as the destination register. With 16 bits, each instruction could only reference one or two registers, so there is some unavoidable overlap between *RS/RD* and *RT/RD*.

Sizes of the operation code (*opcode*) and the function codes (*func* and *bfunc*), were specifically chosen to maximize utilization of the instruction. The 3-bit *opcode* allows for eight values. A single *opcode* is used for all *R-* and *S-Type* instructions, another *opcode* is used for all *B-Type* instructions, and the remaining six *opcode*s are used for *I-Type* instructions. *func* is a 5-bit number used to specify the exact *R-* or *S-Type* instruction. The 5-bit number allows for thirty-two unique instructions. *bfunc* is the 2-bit branch function code. This allows for four different branch instructions while also keeping the *offset* value as large as possible.

The remaining three fields, *shamt*, *offset*, and *immediate*, are all values directly used when executing the instruction. They are the shift amount, address offset, and immediate value respectively.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	opcode			RS				RT/RD				func				
S	opcode			RS/RD				shamt				func				
B	opcode			offset											bfunc	
I	opcode			RS/RD				immediate								

Table 1: Instruction Formats

2.2 Register Type Instructions

R-Type, or Register Type, instructions act on registers, as their name suggests. Each operation utilizes two registers; accessing at least one of the values in the registers and potentially overwriting one of the values in a register. These registers do not need to be unique; for example, you may use the instruction *add r1 r1* to take the value in *r1* twice, add them together, and store the result back in *r1*. Table 2 contains all the *R-Type* instructions along with a short description of their functionality.

2.3 Shift Type Instructions

S-Type, or Shift Type, instructions are a subset of *R-Type* instructions. All *S-Type* instructions utilize the *shamt* field, which is an unsigned 4-bit value directly used as the shift amount operand of the shift operation. Since registers only hold 16-bit values, any type of shift instruction can only significantly shift by up to 16 bits. For that reason, *shamt* is limited to the range zero to fifteen. Any value outside of that range can be equivalently represented with a value inside of the range. The only exception is when the result would always be zero, in which case, another instruction could be used instead. Table 3 contains all the *S-Type* instructions along with a short description of their functionality.

The only shift operations supported are: logical left shift, logical right shift, arithmetic right shift, and rotate right. Arithmetic left shift is functionally equivalent to logical left shift and rotate left can be implemented using rotate right with $16 - shamt$ as the *shamt*.

2.4 Branch Type Instructions

B-Type, or Branch Type, instructions directly modify the program counter. The *offset* field is an address offset from the branch instruction's address. Each branch instruction uses the ALU to add the address in the program counter and the *offset* field's value. Depending on a condition, the program counter is overwritten with the result of this addition. *offset* is a signed, 11-bit value, allowing the branch instruction to jump backward or forward. With an 11-bit value, Branch instructions can jump 1024 instructions backward or 1023 instructions forward. Table 4 contains all the *B-Type* instructions along with a short description of their functionality.

R-Type Instructions	
mov	$RD = RS$
	Copies the value in RS, storing the copied value in RD.
add	$RD = RS + RT$
	Adds the values in RS and RT, storing the sum in RD. May set the *overflow* flag.
sub	$RD = RS - RT$
	Subtracts the value in RT from the value in RS, storing the difference in RD. May set the *overflow*, *less than*, and *zero* flags.
tst	$RS - RT$
	Subtracts the value in RT from the value in RS, discarding the difference. May set the *overflow*, *less than*, and *zero* flags.
and	$RD = RS \,\&\, RT$
	Performs the bit-wise AND operation on the values in RS and RT, storing the result in RD.
or	$RD = RS \mid RT$
	Performs the bit-wise OR operation on the values in RS and RT, storing the result in RD.
xor	$RD = RS \oplus RT$
	Performs the bit-wise XOR operation on the values in RS and RT, storing the result in RD.
st	$[RS] = RD$
	Stores the value in RD at the memory address in RS.
ld	$RD = [RS]$
	Loads the value at the memory address in RS, storing the value in RD. This value cannot be forwarded for the following instruction.

Table 2: Register Type instructions

S-Type Instructions	
sll	$RD = RS << shamt$
	Shifts the value in RS left by *shamt* bits, storing the shifted value in RD. Discards bits shifted too far left and fills empty bits on the right with zeros.
srl	$RD = RS >> shamt$
	Shifts the value in RS right by *shamt* bits, storing the shifted value in RD. Discards bits shifted too far right and fills empty bits on the left with zeros.
sra	$RD = RS >>> shamt$
	Shifts the value in RS right by *shamt* bits, storing the shifted value in RD. Discards bits shifted too far right and fills empty bits with the sign of the original value on the right.
rot	$RD = (RS << (16 - shamt)) + (RS >> shamt)$
	Rotates the value in RS right by *shamt* bits, storing the rotated value in RD. Bits shifted too far right fill the empty bits from the left, one-by-one.

Table 3: Shift Type instructions

2.4.1 Conditional Branching

The conditional branch instructions, *beq* and *blt*, check the flags that are set when the branch (potentially) executes. For this reason, it is recommended to set the flags with the instruction immediately preceding the conditional branch instruction with a *tst* instruction. Savvy programmers could use a *sub* instruction as part of their program logic and control flow, as long as they are careful not to overwrite the flags before the conditional branch instruction.

2.4.2 Branching in the Pipeline

Looking at the architecture (Section 3), *B-Type* instructions are in the *Execute* stage when the branch is (or isn't) taken. Since *Execute* is the third stage, there will already be two additional instructions in the pipeline when the branch is taken. This processor does not discard those instructions. Instead, the two instructions following the branch will always execute. Programmers have the option to insert two *nop* instructions after every branch if they do not want to worry about unusual ordering of instructions or they just want to be safe. To save instructions during execution, programmers could also place useful instructions after the branch. For example, two instructions from the body of the loop could be placed after the conditional branch instruction as they will need to be run every iteration of the loop anyways. Another example is to store the function inputs in the proper registers after a function is called with a *bal* instruction. Note that because of this, the return address of the *bal* instruction is the address after these two instructions.

B-Type Instructions	
b	Unconditionally branches to the destination label.
bal	Unconditionally branches to the destination label. Also saves the program counter when the branch is taken to the return address register.
blt	Conditionally branches to the destination label only when the *sign* flag is set. Otherwise, the instruction has no effect.
beq	Conditionally branches to the destination label only when the *zero* flag is set. Otherwise, the instruction has no effect.

Table 4: Branch Type instructions

```
mov   r0  r1
addi  r1  0x0ff
sll   r1  #8
addi  r1  0x0ff
```

Figure 1: A sequence of instructions to use a large immediate value

2.5 Immediate Type instructions

I-Type, or Immediate Type, instructions perform similar operations to *R-Type* instructions, except one of the values comes from the *immediate* field in the instruction. This 9-bit *immediate* field is directly used as the value in the operation and is sign-extended. *immediate* values range from -256 to 255.

To make the *immediate* field as large as possible, *I-Type* instructions have no function code. This means that each *opcode* for Immediate Type instructions is unique. It also means that the number of *I-Type* instructions is very limited, in this case to six. This architecture only implemented two, with room for four more. *add* and *and* were deemed as the most common operations to need a specific value for and, therefore, were implemented.

Conveniently, the *immediate* field can be represented with a sign bit and two hexadecimal digits. A simple sequence of operations to effectively get a 16-bit immediate is seen in Figure 1.

I-Type Instructions	
addi	$RD = RS + immediate$
	Adds the value in RS with the sign-extended immediate value, storing the sum in RD. May set the *overflow* flag.
andi	$RD = RS \,\&\, immediate$
	Performs the bit-wise AND operation on the value in RS with the sign-extended immediate value, storing the result in RD.

Table 5: Immediate Type instructions

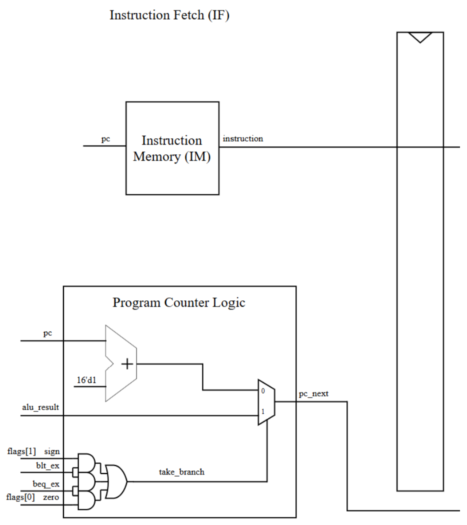

Figure 2: The Instruction Fetch (IF) stage of the architecture

3 Architecture

Similar to the *MIPS* [1] architecture, this processor implements a 5-stage pipeline. Each instruction will complete one stage sequentially every clock cycle beginning with Stage 1 and the following instruction beginning when the previous instruction reaches Stage 2. In the middle of executing a program, five instructions will be processed simultaneously, each in a different stage.

This processor was designed using structural Verilog at the top level and behavioral Verilog for the components. Between each of the stages is a pipelining unit which only allows data to be propagated on the positive edge of the clock cycle.

The processor utilizes the internal Basys3 [2] 100 MHz clock as its clock.

3.1 Stage 1: Instruction Fetch

Instruction Fetch (IF) is a simple stage with two components: *Instruction Memory (IM)* and *Program Counter Logic*. The datapath of the *IF* stage is shown in Figure 2.

3.1.1 Instruction Memory

IM simply takes an address as its input and outputs the 16-bit value at that address. In this case, the address is the *Program Counter (PC)* which is the address of the instruction to begin processing. The instruction output from *IM*

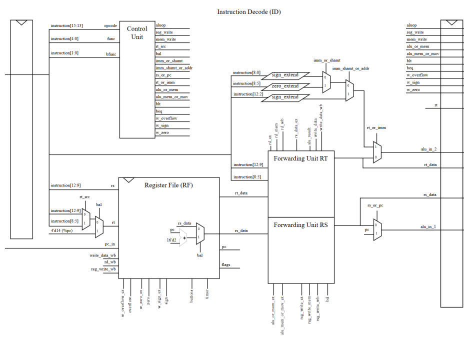

Figure 3: The Instruction Decode (ID) stage of the architecture

is the only signal to be propagated to the next stage. *Instruction Memory* is word addressable and disjoint from all other memory.

3.1.2 Program Counter Logic

Program Counter Logic is a component that, in most scenarios, only adds 1 to the *PC*. This simple functionality is how the instructions in *IM* are executed sequentially. The *Program Counter Logic* component also takes inputs from other stages. Specifically, it takes control data and flags along with the result of the *ALU* operation. It processes the control data and flags to determine whether a branch is being taken this cycle. If it is, the *ALU* result is used as the new *PC* value.

3.2 Stage 2: Instruction Decode

Instruction Decode (ID) is a more complex stage. At the beginning of the clock cycle, *ID* takes the instruction to be executed from *IF*. Based on the instruction, it sets some control data and retrieves values from the *Register File*. The main components of this stage are the *Control Unit (CU)* and the *Register File (RF)*. This section also contains the important *Data Forwarding Unit*. The datapath of the *ID* stage is shown in Figure 3.

3.2.1 Control Unit

The *CU* effectively determines which instruction is being processed. Based on the *opcode* and potentially the *func* and *bfunc*, many control data and flags are set. These are exclusively used to select data. For example, the *aluop* selects the operation that the *ALU* will perform. This data is used in *ID* and all successive stages.

Registers	
r0	Always holds the value zero and cannot be written to.
r1-r10	General purpose registers for regular program operation.
r11	Holds the Stack Pointer. The stack is full-descending and the stack pointer is initialized to the end of data memory.
r12	Holds the Global Pointer. Initialized to the start of regular data memory.
r13	Holds the Return Address. Automatically written to by *bal* instructions. Must be saved elsewhere during nested function calls.
r14	Holds the Program Counter. Automatically written every clock cycle with the address of the next instruction to be processed.
r15	Holds the Flags. The proper flags are set/reset or unmodified every clock cycle.

Table 6: Register Purposes

3.2.2 Register File

The *RF* is where all of the registers and their associated data are stored. The instruction being processed will read from the *Register File* during *ID* and will write back to the *RF* during the *Write Back* stage. The registers are numbered, zero through fifteen and each has a dedicated purpose described in Table 6.

3.2.2.1 Flags

There are three flags automatically set or reset by specific instructions. Bit 0 is the *zero* flag, bit 1 is the *less than* flag, and bit 2 is the *overflow* flag. The next five bits correspond to button presses, covered in detail in Section 3.6.2. Bit 8 is the timer flag, covered in detail in Section 3.6.3.

3.2.3 Data Forwarding Unit

The *Data Forwarding Unit* is one of the most important parts of a pipelined processor. Registers are written to at the end of the *Write Back* stage, which is Stage 5. Instructions retrieve data from the *RF* here in the *ID* stage, which is Stage 2. Without forwarding, there would need to be a delay of three instructions between the instruction that writes to a register and the instruction that reads from that same register. Since most values to be written are found during the *Execute* stage, which is Stage 3, that value can be found and forwarded to the following instruction before it is needed. During a *ld* instruction, data is retrieved from *Data Memory* in the *Memory* stage, which is Stage 4. This data can also be forwarded, but there is a necessary delay of one instruction before it can be used. This data, along with control data, from each of the three remaining stages is used to determine if data can be forwarded to the instruction

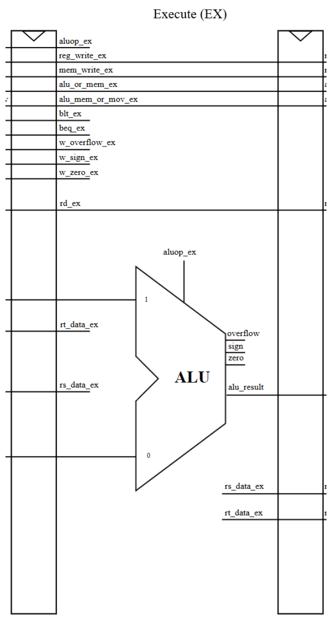

Figure 4: The Execute (EX) stage of the architecture

in the *ID* stage. If the data can and should be forwarded, it overwrites the data read from the *RF*.

3.3 Stage 3: Execute

Execute (EX), despite being where instructions are actually executed, is just the *ALU*. The datapath of the *EX* stage is shown in Figure 4.

3.3.1 Arithmetic and Logic Unit

The *Arithmetic and Logic Unit (ALU)* takes two values from *ID* and performs an operation on them. The result is propagated to the next stage. All but three instructions utilize the *ALU*: *mov*, *st*, and *ld*. Branch instructions use the *ALU* to add the *offset* and the *PC* to use as the branch target. All other instructions simply use the *ALU* for their designated operation.

3.4 Stage 4: Memory Access

Memory (MEM) only contains the *Data Memory* component. The datapath of the *MEM* stage is shown in Figure 5.

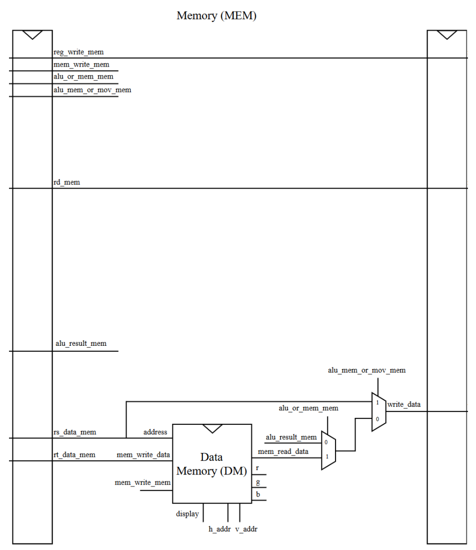

Figure 5: The Memory (MEM) stage of the architecture

3.4.1 Data Memory

DM is only utilized by two instructions: *st* and *ld*. These two instructions are used very often as most programs will need more than the ten general purpose registers this architecture provides. In addition, the stack is located in data memory and, as will be seen in Section 3.6.1, the VGA display array is stored in *DM*. Due to hardware limitations, *DM* had to be kept small, in this case 1500 words. This is also why *DM* is word-addressable, the limitation is from how many addresses there are. Making memory byte-addressable would double the number of addresses or halve the potential size.

3.5 Stage 5: Write Back

WB is a small stage. It exists solely to finalize the data to be written back to the *Register File* at the end of the cycle. Due to design changes late in development, some components of the *WB* stage were moved to other stages. In its current state, writing to registers could instead occur at the end of the *MEM* stage, removing the fifth stage entirely. The datapath of the *WB* stage is shown in Figure 6.

3.6 Peripheral Units

The peripheral units of the architecture include a *VGA Controller*, *buttons*, and a *timer*. The datapath of the peripherals is shown in Figure 7.

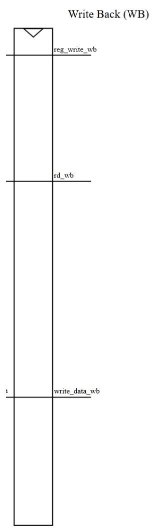

Figure 6: The Write Back (WB) stage of the architecture

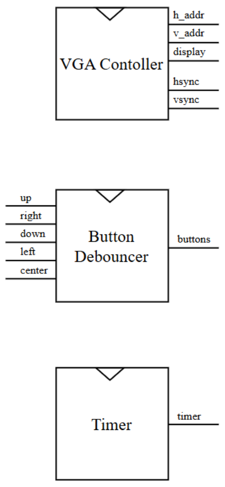

Figure 7: The Peripherals of the architecture

3.6.1 VGA Controller

The Video Graphics Array outputs to a 640x480 display. Each pixel is represented with a 12-bit RGB value, with 4 bits for each of red, green, and blue. The pixels are individually stored in *Data Memory* sequentially beginning at address 0. This allows the programmer to directly modify the pixels being displayed. Due to the hardware limitations, the display was converted to an effective 40x30 pixels. This was done by keeping the same horizontal and vertical counters to select a pixel to display, but shifting each counter right by four bits to go from a maximum of 639 and 479 to 39 and 29. The display is still 640x480 pixels, but the same memory location is used for 16 consecutive pixels and the same row is displayed 16 consecutive times. The *VGA Controller* follows the standard VGA timing [3] for a 640x480 pixel display. This controller divides the internal Basys3 [2] 100 MHz clock by four for a 60 Hz refresh rate.

3.6.2 Button Inputs

All of the buttons are debounced before they are used by the processor. The Basys3 FPGA [2] has five buttons utilized by this architecture. They are internally referred to as: *up_btn*, *right_btn*, *down_btn*, *left_btn*, and *center_btn*. Each has a corresponding flag in the same order from bits 7 to 3 of the flags register. On the positive edge of the debounced button input, the corresponding flag is set. These buttons are not interrupts, instead they require designated polling in software. In addition, the flags will need to be reset in software if they are ever recognized as set and processed.

3.6.3 Timer

The timer, similar to the buttons, sets a flag when the timer is up. In this architecture, it is a simple counter that increments every clock cycle. When the timer is up, the flag is set, the counter is reset, and the timer starts again. Also similar to the buttons, the flag requires polling in the software to be recognized and the flag will need to be manually reset.

4 Assembler

The assembler was written in 150 lines of Python. It is a mostly straightforward assembler which parses the *.s* assembly files to create *.mem* memory files for the processor to read. It converts each instruction to binary following the instruction format. The assembler also has a few additional features to make writing the assembly code a little bit easier. The assembler assumes the assembly code is well-formed and may not return an error if it is not well-formed.

4.1 Pseudo-Instructions

Pseudo-Instructions are additional common instructions that can be easily implemented using already present functionality.

4.1.1 No Operation (nop)

A *nop* is a simple instruction that is used to stall. Specifically, the instruction should have no effects while being processed. Since register *r0* is always zero, a *nop* is implemented as *mov r0 r0*.

4.1.2 Return (ret)

A *ret* instruction has a simple purpose: to return program execution to where the currently executing function was called from. In other words, the *Program Counter* should be set to the *Return Address*. This can be implemented as *mov ra pc*. As a note, since the *Program Counter* will not be updated until the end of the *WB* stage, there will be four instructions already in the pipeline before the function returns. The four instructions following a *ret* will always be executed. As most functions will need to *push* multiple registers to the stack and *pop* them at the end of the function call, Simply *pop*ping two of the registers after the *ret* will waste no instructions.

4.1.3 Push and Pop

push and *pop* instructions can be written as two consecutive instructions for each register value. These are common instructions that utilize the stack. Since the stack is full-descending in this implementation, a *push* instruction will first decrement the *Stack Pointer*, then it will store a register value in memory at the *Stack Pointer's* address. A *pop* instruction will do the opposite: load the value from memory then increment the *Stack Pointer*. This works especially well if the value *pop*ped is needed in the following instructions, as the necessary delay after reading from memory will be filled with adjusting the stack pointer. Each of these instructions can also *push* or *pop* a list of registers. *push ra r1* will *push* the registers in that order: *ra* then *r1*. *pop ra r1* will *pop* the registers in the reverse order: *r1* then *ra*.

4.2 Registers

The function *regToBin()* takes a register name in the form of a string and returns the corresponding register number in binary, also as a string. This string is included in any instructions that use that register. The function is also utilized when converting a hexadecimal digit to its corresponding binary value. The various names of registers can be seen in Table 7.

Binary	Name				
0000	0	r0	%0	zero	%zero
0001	1	r1	%1	gp1	%gp1
0010	2	r2	%2	gp2	%gp2
0011	3	r3	%3	gp3	%gp3
0100	4	r4	%4	gp4	%gp4
0101	5	r5	%5	gp5	%gp5
0110	6	r6	%6	gp6	%gp6
0111	7	r7	%7	gp7	%gp7
1000	8	r8	%8	gp8	%gp8
1001	9	r9	%9	gp9	%gp9
1010	10	r10	%10	gp10	%gp10
1011	11	r11	%11	sp	%sp
1100	12	r12	%12	gp	%gp
1101	13	r13	%13	ra	%ra
1110	14	r14	%14	pc	%pc
1111	15	r15	%15	flags	%flags

Table 7: Register Names

4.3 Constants

Constant immediate values are also automatically converted from decimal or hexadecimal to binary. In addition, binary values are sign-extended to 9 bits. The function *immToBin()* converts any immediate value with a preceding # from decimal to 9-bit binary, a preceding *0x* from hexadecimal to 9-bit binary, or a preceding *0b* from binary to sign-extended 9-bit binary. Decimal and hexadecimal values can also be signed.

4.4 Branch Labels

Branch labels are the most complex part of the assembler. Since branch instructions branch based on an *offset* from the current instruction, modifying the assembly code also means this *offset* value will be modified. The assembler first parses through the entire program, keeping track of where all the labels are. In the second pass, the assembler actually converts the assembly instructions into binary. Now, since the assembler knows where all the labels are in the program, it can calculate the offset from each branch instruction to the label. In each pass, the assembler also accounts for pseudo-instructions that are made up of more than one instruction.

```
main_loop:
    mov flags r6
    andi r6 0b100000000
    wait_for_timer:
    tst r0 r6
    beq wait_for_timer
    mov flags r6
    andi r6 0b100000000
```

Figure 8: The main loop of the program; waiting for the timer to go off

5 Using the Processor

A version of *Atari Breakout* [4] was developed using the custom assembly language described in Section 2 for the custom architecture detailed in Section 3. With comments, the written program was 530 lines of assembly. After generating the machine code, the program was 400 binary instructions.

5.1 Breakout

A game of *Atari Breakout* [4] contains a few key components. A constantly moving ball, a player controlled paddle, and an array of blocks. The goal of the game is to use the paddle to redirect the ball and destroy each of the blocks.

The program flow from the start of the program is as follows. First, the game state is initialized. This includes clearing the screen, placing each of the blocks in the array, and setting both the ball and paddle to their initial positions. The game can also be reset by branching to this location.

Then, the game waits for the timer flag to be set. This is a simple, short loop shown in Figure 8. Note the two instructions after the *beq* are part of the loop body. Also note that if the branch is not taken, the *Program Counter* will just continue to be incremented. If the timer does go off, the next lines of code will handle what happens next.

Next is the *next_frame* section of the code. Every time the timer goes off, the program will check for any pressed buttons. It will handle their processes accordingly before moving on to updating the game state. The paddle will first be redrawn followed by the ball. When the ball moves, any blocks hit will be destroyed. If the ball hits anything, the direction of the ball will also be updated accordingly.

5.2 Functions

A total of seven helper functions were written for the game.

modulo finds the remainder of a division, this is used to determine the location of a pixel in a row or to determine the leftmost address of a hit block.

reset_screen and *generate_blocks* are both used in initializing or resetting the game. They clear every pixel on the screen and place the block array respectively.

store_block is a function with many uses. Given an address and a color, it will set the five consecutive pixels beginning with the given address to the given color. In addition to being used to place a block, it can also be used to clear a block and to move the paddle. Both the paddle and each block are one by five pixels.

move_paddle is a fairly straightforward function that takes the updated position of the paddle each frame (based on buttons pressed). It sets the color of all five pixels of the paddle to white and clears the pixels on either side. Since the paddle can only move one pixel in either direction at a time, this will clear the previous paddle and place the new one. Also, since this function is called every frame before the ball is moved, even if the ball pixel is cleared by this function, it will quickly be replaced by moving the ball.

move_ball is the bulk of the program, taking about 200 of the lines of assembly. This contains all of the logic for how a ball moves. Since the ball can move in one of eight directions, there are a lot of possibilities. The ball also might hit a number of other game objects which will each change how the ball moves. This function accounts for the ball hitting any block, the paddle, or the wall and ceiling. It also accounts for the ball falling off of the screen. Any blocks it hit are also cleared.

6 Acknowledgments

I would like to thank a few people who helped me throughout my time as a student at the University at Buffalo. First are two of my professors, Dr. Kris Schindler and Dr. Ramalingam Sridhar. Both of these professors sparked my interest in computer architecture and hardware through their classes and they were always willing to help when I needed it. I would also like to thank my undergraduate Computer Engineering class of 2024 along with one of our TAs, Anthony Roberts. I would not have been nearly as involved in the Computer Engineering program if it wasn't for their encouragement.

References

[1] Patterson, David A., and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* 4th ed., Morgan Kaufmann, 2012.

[2] Basys 3 reference manual - digilent reference. https://digilent.com/reference/programmable-logic/basys-3/reference-manual

[3] VGA signal 640 x 480 @ 60 hz industry standard timing. http://www.tinyvga.com/vga-timing/640x480@60Hz

[4] Breakout. Atari®. https://atari.com/pages/breakout