LY86-64: Implementation and Evaluation of a Y86 Browser-Based Simulator

By

Chau Bao Ly

Honors Thesis

Appalachian State University

Submitted to the Department of Computer Science

in partial fulfillment of the requirements for the degree of

Bachelor of Science

May 2021

Approved by:

_____

Cindy Norris, Ph.D., Thesis Project Director

_____

Frank Barry, M.S., Second Reader

_____

Alice McRae, Ph.D., Departmental Honors Director

_____

Raghuveer Mohan, Ph.D., Departmental Honors Director

Abstract

The Y86-64 PIPE project is a seminal project for Appalachian State University Computer Science majors. For many students, it is their first software product of significant size. After completing the project, students have a better understanding of computer systems and the software engineering skills and tools needed to develop large pieces of software. However, to implement the code, students need a sophisticated understanding of the machine being simulated. Of particular difficulty is the control logic and signals that direct the stages of the PIPE machine. Several Y86-64 simulators are available, but existing simulators display only the contents of memory and general-purpose registers. They do not provide a visualization of the complicated control logic and signals applied to pipeline registers. For this reason, we undertook the development of the LY86-64 (pronounced “lee 86-64”), a Y86-64 browser-based simulator. A survey of 47 students, some currently studying the Y86-64 PIPE machine and some who studied the machine in a previous semester, found that 100% of respondents believed that the simulator provides a better understanding of the control logic.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

Software developers who have studied computer systems are better prepared to find and correct bugs and write code that is fast and efficient. For example, an understanding of the cache memory can enable a programmer to write code which uses the cache effectively, i.e., has good locality. Programmers that understand virtual memory also know what a segmentation fault is. Incorrect program results could be due to an overflow or finite precision arithmetic, both which are artifacts of computer hardware. Understanding instruction-level parallel architectures allows a programmer to consider the control and data hazards that can impact the performance of the code. In general, a comprehension of computer systems can enable developers to write correct, secure, and efficient code [4].

Unfortunately, computer systems courses are notoriously difficult. The three-semester sequence of systems courses required for Computer Science majors at Appalachian State University has the reputation of being among the most challenging required courses for the major. For this reason, educators have investigated numerous techniques for making the material more accessible, including the use of architecture simulators [1, 2, 9, 12, 13, 14, 17, 18, 19]. Of particular interest are the simulators developed to aid students in understanding computer architecture.

At Appalachian State University, Computer Science majors implement a Y86-64 PIPE project, a seminal project for the Computer Systems 1 course. For many students, it is their

first software product of significant size. After completing the project, students have a better understanding of computer systems and the software engineering skills and tools needed to develop large pieces of software. However, to implement the code, students need a sophisticated understanding of the machine being simulated. Of particular difficulty is the control logic and signals that direct the stages of the PIPE machine. Several Y86-64 simulators are available, but existing simulators display only the contents of memory and general-purpose registers. They do not provide a visualization of the complicated control logic and signals applied to pipeline registers. For this reason, we undertook the development of the LY86-64 (pronounced “lee 86-64”), a Y86-64 browser-based simulator.

The LY86-64 simulator focuses on providing students with a visualization of control logic and signals, specifically stalling and bubbling. LY86-64 supports 64-bit signed integer operations and is based on the implementation of the PIPE architecture in Chapter 4 of the textbook *Computer Systems: A Programmer’s Perspective* by Randal E. Bryant and David R. O’Hallaron [4].

The remainder of this thesis is organized as follows:

- Chapter 2 covers the required background information. Specifically, the background information covers Y86-64 and Angular.
- Chapter 3 discusses related work. Five other Y86 simulators are described.
- Chapter 4 describes the implementation of the LY86-64 simulator in detail.
- Chapter 5 provides the results of surveys given to students who used the LY86-64 simulator.
- Chapter 6 covers plans for the future.

Chapter 2: Background

In this chapter, we provide the information needed to understand the LY86-64 simulator. Section 2.1 explores the Y86-64 PIPE Machine, its instruction set, and different features. Subsequently, in Section 2.2, we introduce Angular, its architecture, and concepts.

2.1 The Y86-64 PIPE Machine

The textbook *Computer Systems: A Programmer's Perspective* by Randal E. Bryant and David R. O'Hallaron [4] provides an overview of the computer systems content that programmers need to understand to write correct, secure and high-performance code. Chapter 4 of the textbook focuses on pipeline architectures. Programmers can write higher performance codes if they understand that dependencies among instructions can impact modern, instruction-level parallel architectures. Pipelined machines are the basis of these architectures. The book introduces pipelined machines by starting with a sequential architecture and making incremental changes that lead to pipelining. Specifically, the chapter describes four architectures that implement the Y86-64: SEQ, SEQ+, PIPE-, and PIPE. The PIPE architecture is the basis of the LY86-64.

2.1.1 Instruction Set

The Y86-64 is based on the X86-64 instruction set architecture. However, the Y86-64 has much fewer operations and addressing modes. The Y86-64 does not include floating-point instructions, and all operations use 8-byte signed integer operands. The addressing modes supported include displacement (also known as base plus offset), register, immediate, and register indirect (base plus offset with an offset of 0). Not all addressing modes can be used with all instructions. In the Y86, the addressing modes used for the source and destination operands are implied by the opcode.

Figure 2.1 displays a chart of the register identifiers for the Y86-64 processor's register file. There are a total of fifteen registers numbered zero to fourteen. By convention, several registers are reserved for particular purposes. For example, `%rsp` contains the pointer to the stack, and instructions like `popq` and `pushq` make changes to `%rsp` by incrementing and decrementing it. In a function with several parameters, `%rdi`, `%rsi`, and `%rdx` are used for the first, second, and third arguments. The register `%rax` is reserved for the return value of a method.

Number	Register name	Number	Register name
0	%rax	8	%r8
1	%rcx	9	%r9
2	%rdx	A	%r10
3	%rbx	B	%r11
4	%rsp	C	%r12
5	%rbp	D	%r13
6	%rsi	E	%r14
7	%rdi	F	No register

Figure 2.1 Y86-64 register identifiers

Figure 2.2 shows the Y86 instruction set. There are a total of twenty-seven instructions, including four moves, four arithmetic operations, seven jumps, six conditional moves, halt, nop, call, return, push, and pop. The four move instructions are `irmovq`, `rrmovq`, `mrmovq`, and `rmmovq`. The first two letters in the move instructions indicate the source and destination, respectively. The source is either an immediate value (`i`), register value (`r`), or a value from memory (`m`), and the destination is either a register (`r`) or a memory (`m`) location. Operands in memory are accessed using displacement mode. For example, the destination of the `rmmovq` instruction in Figure 2.2 is obtained by adding the value of the register `rB` and the value of `D` together. `rB` represents the base register, and `D` is the displacement.

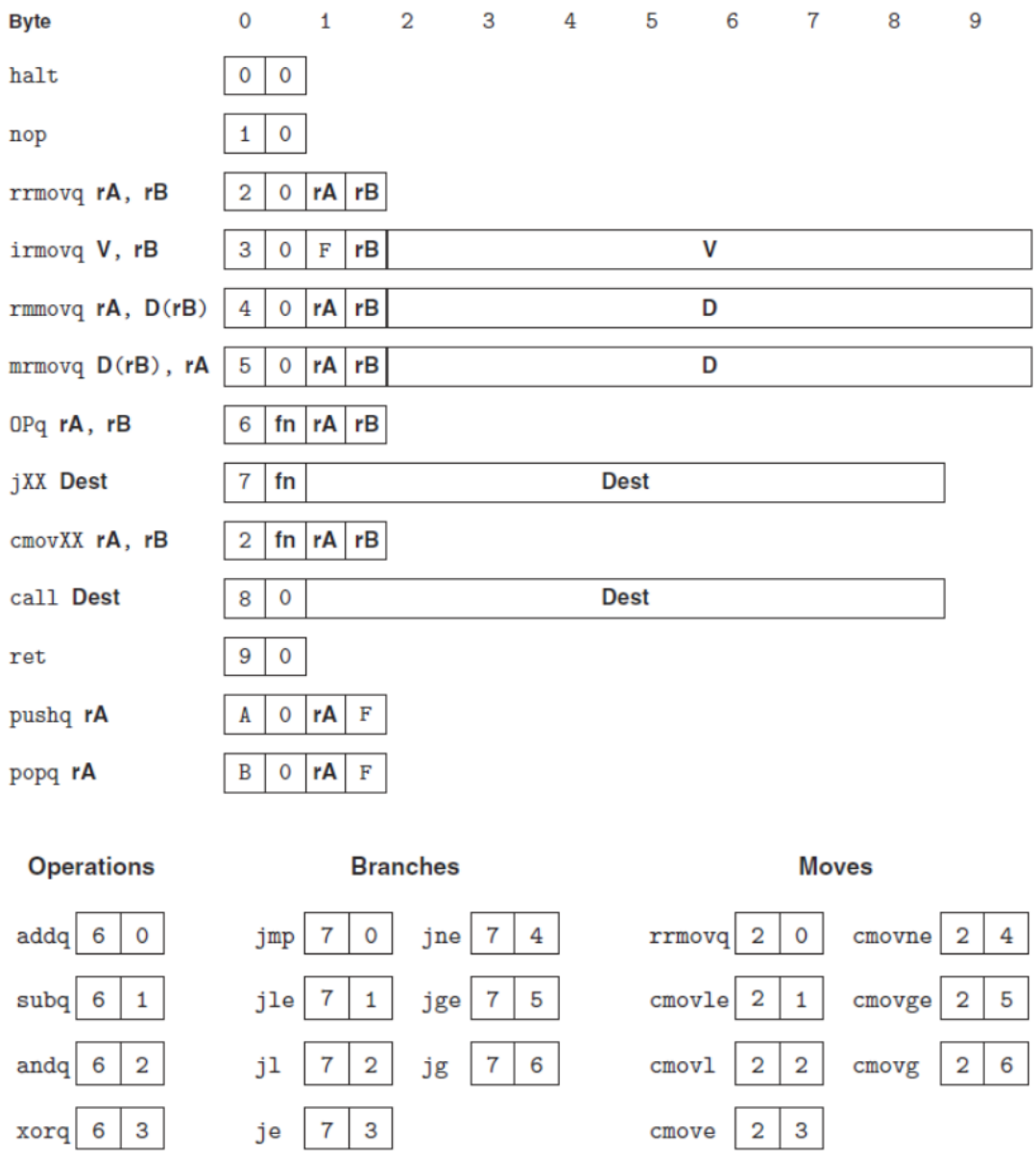

Figure 2.2 Y86-64 instruction set

Instructions for arithmetic operations (`OPq`) include `addq` for addition, `subq` for subtraction, `andq` for bitwise AND, and `xorq` for bitwise exclusive OR. Register `rB` in the `OPq` instruction in Figure 2.2 is both a source and destination register. For example, in the

case of `addq %rax, %rcx`, the values of registers `%rax` and `%rcx` are added together and the result is stored in the register `%rcx`. Condition codes (flags) are also set based on the result. Specifically, the zero flag (ZF), sign flag (SF) and overflow flag (OF) are set as follows:

```
ZF = (result == 0);
SF = (result < 0);
OF = ((operand1 < 0) && (operand2 < 0) && (result > 0)) ||
     ((operand1 > 0) && (operand2 > 0) && (result < 0));
```

An OPq instruction typically precedes jump instructions. The `OPq` instruction sets the flags, and the jump is taken or not based upon the values of the flags. The jump instructions, `jle`, `jl`, `je`, `jne`, `jge`, and `jg`, determine whether to execute the jump or not based on the condition codes. The letters after the `j` specify the condition; `l` is for less than, `e` is for equal, and `g` is for greater. For example, a `jl` is taken if the result of the most recently executed `OPq` is less than zero, i.e., SF is 1 and OF is 0 or SF is 0 and OF is 1. Notice that the jump is taken (or not) based upon the sign of the arithmetic result (assuming infinite precision) and not the sign of the result in the destination register.

The instructions `cmovle`, `cmovl`, `cmove`, `cmovne`, `cmovge`, and `cmovg` make up the conditional move instructions in the Y86-64 Instruction Set Architecture (ISA). The letters `cmov` indicate that the instruction is a conditional move. The letters following `cmov` express the condition to be evaluated. Conditional moves only perform the register to register move when a condition is true. For example, the instruction `cmovg %r9, %rax` sets `%rax` to

the value of `%r9` if the result of the most recent `OPq` is greater than 0, which is indicated by `ZF == 0` and `SF == OF`. The `rrmovq` is an unconditional move; the move is performed regardless of the condition codes.

The `call`, `ret`, `pushq`, and `popq` instructions support procedure calls. The `call` instruction pushes the return address onto the stack and jumps to the destination address, and the `ret` instruction pops the return address from the stack and jumps to that address. Similarly, the `pushq` and `popq` instructions perform push and pop operations to the stack. These instructions can be used to perform callee or caller saves and restores.

Finally, the `halt` instruction tells the machine to stop executing program instructions, and the `nop` instruction injects a software bubble, which does nothing (no operation).

The program below demonstrates a maximum function that compares two numbers in memory and writes the greater of the two values to the memory location following the two numbers.

```
.pos 0
            irmovq nums, %rsi
            mrmovq (%rsi), %rax
            mrmovq 8(%rsi), %rdi
            rrmovq %rdi, %rbx
            subq %rax, %rdi
            cmovg %rbx, %rax
            rmmovq %rax, 16(%rsi)
```

```
            halt
            .align 8
nums:       .quad 20
            .quad 23
            .quad 0
```

The first instruction, `irmovq nums, %rsi` loads the address corresponding to label `nums` into register `%rsi`. Specifically, the label nums is a symbol for the address of the value 20 stored in a quadword (eight bytes) of memory. The next two instructions, `mrmovq (%rsi), %rax` and `mrmovq 8(%rsi), %rdi` are memory to register move instructions which load 20 and 23 into `%rax` and `%rdi`, respectively. The next instruction, `rrmovq %rdi, %rbx`, performs a register-to-register move to copy the value 23 from register `%rdi` into register `%rbx`. This is needed because the next instruction, `subq %rax, %rdi`, overwrites register `%rdi`. The instruction `subq %rax, %rdi` performs the following operation: `%rdi = %rdi - %rax`. When the `subq` instruction is executed, the condition flags are set. In this case, since 23 (stored in `%rdi`) is greater than 20 (stored in `%rax`) the flags are set as follows: `SF = 0, OF = 0, and ZF = 0`. The flags indicate that the result of the subtraction is a positive number. The next instruction, `cmovg %rbx, %rax`, is a conditional move. It will only perform this register-to-register move if the condition flags indicate that the previous instruction results are positive. Thus, the value stored inside of `%rbx` is then copied into `%rax` in this case. The `rmmovq` instruction is then used to store the value of `%rax` into the memory location that follows the quadword holding 23. The `halt` instruction indicates the end of the program.

2.1.2 PIPE Machine

The textbook presents the PIPE machine by starting with a purely sequential architecture known as SEQ and making modifications to transform it into a five-stage pipeline processor. The SEQ machine describes a complete sequential processor that performs the execution of an instruction across six stages: fetch, decode, execute, memory, writeback, and PC update. Because a pipelined machine must calculate a PC value every clock cycle, the first modification to the SEQ architecture is to rearrange the computational stages to calculate and select the PC before the fetch. This resulting hardware is known as SEQ+. The SEQ+ processor uses state registers to store computed values before updating the program counter (PC) on the new clock cycle. The SEQ processor, on the other hand, calculates the PC using the values computed in the same clock cycle.

The PIPE− hardware is a pipelined version of SEQ+ hardware derived by inserting a pipe register before each computational stage. The PIPE− has five stages: fetch, decode, execute, memory, and writeback. The pipeline registers F, D, E, M, and W are used to provide input to these stages. Stages of the pipelined machine operate in lockstep on different instructions; thus, five instructions can be overlapped in execution. The PIPE− and PIPE are almost identical to each other. However, the PIPE processor contains additional logic to prevent pipeline hazards. The PIPE− machine relies on the compiler writer or Y86-64 programmer to prevent hazards by inserting nops in the code.

Figure 2.3 shows the hardware structure of the PIPE machine. The remainder of this section will focus on the PIPE machine since this machine is the basis of the LY86-64 simulator.

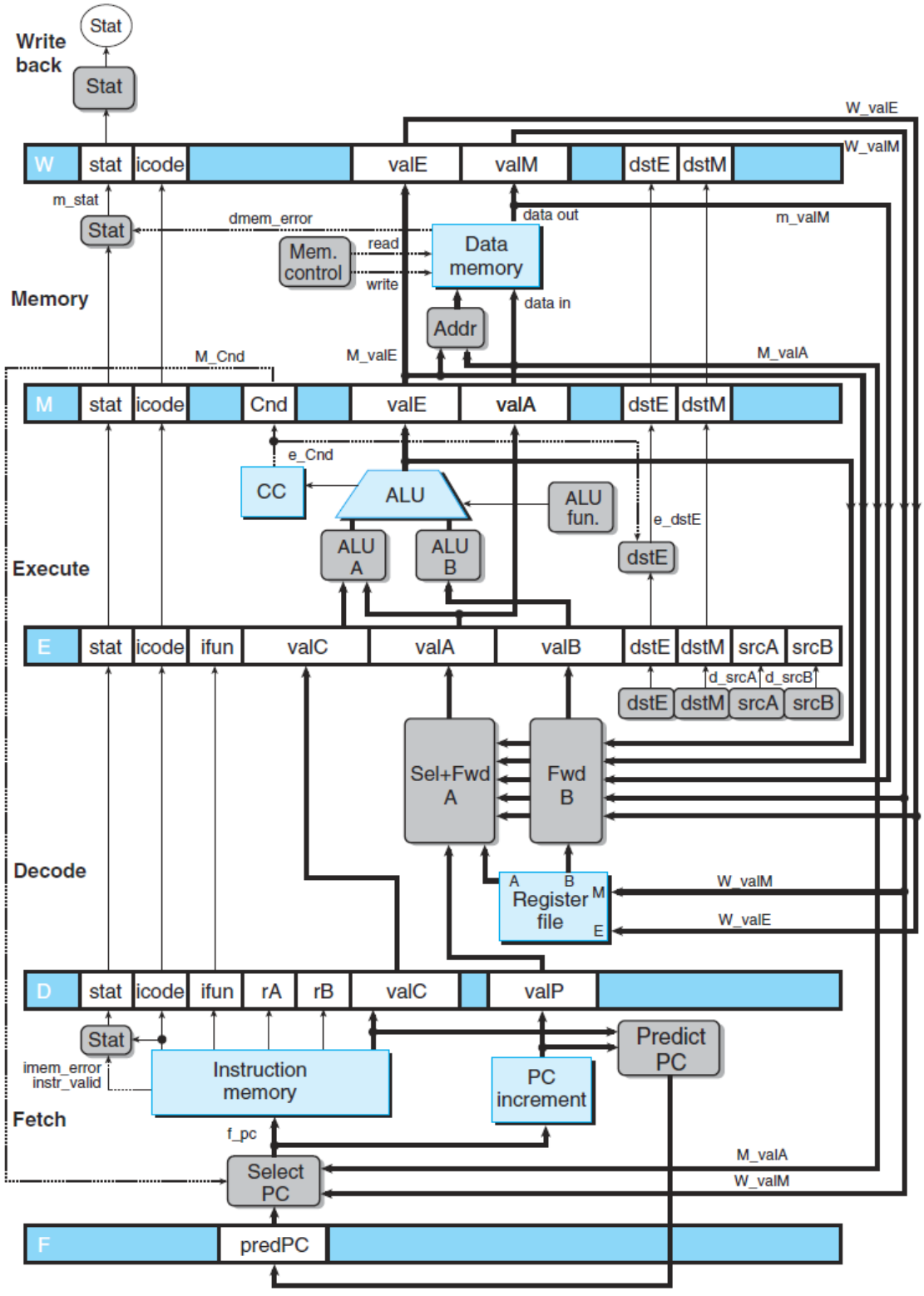

Figure 2.3 Y86-64 PIPE machine's hardware structure

Fetch Stage and the F Pipeline Register

In the fetch stage, the `SelectPC` unit selects the PC by choosing either the address of a fallthrough instruction from a mispredicted branch (`M_valA`), the return address (`W_valM`), or the predicted PC from the previous cycle's instruction (`predPC`). After the PC value is selected, the processor fetches the instruction in memory at the location `f_pc`. The `Stat` unit calculates the status of the fetch to be either: OK, halt fetched, bad address, or invalid opcode. Using the selected PC and the fetched instruction, the `PC increment` unit calculates the following sequential instruction address (`valP)`. The `Predict PC` unit calculates the value for `predPC`, which is the predicted PC value for the next fetch. If the instruction is a jump or a `call`, the unit predicts the next PC value to be `valC`, which is the target of the jump or `call`. Otherwise, the `Predict PC` unit selects the value of `valP` for `predPC`.

Decode Stage and the D Pipeline Register

The D pipeline register resides between the fetch stage and the decode stage to store the output of the fetch stage, which is then inputted to the decode stage in the next cycle. As can be seen in Figure 2.3, the D pipeline register contains the values for `stat`, `icode`, `ifun`, `rA`, `rB`, `valC`, and `valP`. `icode`, `ifun`, `rA`, `rB`, and `valC` are obtained via the instruction that was last fetched. These fields are identified in Figure 2.2. If the fetched instruction identifies fewer than two registers, then `rA` and/or `rB` will be set to 0xf (no register). For example, for the `nop` and `halt` instructions, `rA` and `rB` are both set to 0xf.

valC is set to the value identified as V, D, or Dest in Figure 2.3, or 0 if the instruction does not contain an immediate value. valP contains the address of the next sequential instruction. As described earlier, stat contains the status of the fetch. The decode stage mainly determines values for valA and valB based on the instruction. For valA, the Select + Fwd A unit chooses either valP from the D pipeline register, the value from the A port of the register file, or values from the five forwarding sources from the later pipeline stages. On the other hand, the decode stage does not use valP to determine valB. It uses either value from the B port of the register file or values from the forwarding sources. The topic of forwarding will be discussed in Section 2.2.1. In addition to choosing the values for valA and valB, the decode stage determines the source registers for the A and B ports and destination registers for the E and M ports.

Execute Stage and the E Pipeline Register

Similar to the D pipeline register, the E pipeline register is located between the decode stage and the execute stage. The E pipeline register contains values produced by the decode stage that are then used as input for the execute stage. Outputted values from the decode stage are stat, icode, ifun, valC, valA, valB, dstE, dstM, srcA, and srcB.

In the execute stage, the most important component is the Arithmetic/Logic Unit (ALU) which performs either addition, subtraction, exclusive-or, or bitwise *and* operation depending upon the opcode. The component labeled alufun determines which operation to perform. For OPq instructions, the fn field (see Figure 2.3) identifies the operation to perform. For every other instruction, the alufun unit generates the signal that causes an

add operation to be performed. Thus, even for non-OPq instructions, the ALU will produce an output. For example, in the case of a nop, the ALU will calculate 0 + 0. The inputs to the ALU are provided by the two multiplexers labeled aluA and aluB. Depending on the instruction, the value selected by aluA could be valA, valC, +8, or -8. For aluB, the value could be either valB or 0. For example, for a popq instruction, the value selected by aluA would be 8; the value selected by aluB would be the value of the stack pointer. If the instruction in the icode field of the E register is an OPq, the unit labeled CC will calculate the condition codes. If the instruction in the E register is a jXX or a cmovXX, a cond unit (not identified in Figure 2.3) will use the current values of the condition codes and the instruction's function code (ifun) to determine whether the jump is taken or the conditional move is performed. The output of cond, 0 or 1, is stored in the M register field labeled Cnd.

Memory Stage and the M Pipeline Register

The M pipeline register is in between the execute stage and the memory stage. The pipeline register contains values for stat, icode, Cnd, valA, dstE, dstM, and the valE calculated by the ALU.

Some Y86-64 instructions either explicitly or implicitly cause memory to be accessed. For example, the instructions mrmovq and rmmovq cause memory to be read from and written to, respectively. In addition, popq, pushq, call and ret cause memory to be read from (popq and ret) and written to (pushq and call). These memory operations are performed during the memory stage. If the memory control unit indicates

memory read or write, the processor will access data memory. For read operations, `valM` stores the value read from memory. For write operations, the processor will write `valA` to the address specified by the instruction. Similar to the fetch stage, the memory stage can change the `stat` field if there is an error when reading and/or writing to memory. Section 2.2.1 discusses how the PIPE machine handles exceptions in greater detail.

Writeback Stage and the W Pipeline Register

The W pipeline register between the memory stage and the writeback holds the values for `stat`, `icode`, `valE`, `valM`, `dstE`, and `dstM` produced by the execute stage. The writeback stage is the last stage in the pipeline. The writeback stage writes the value of valE and valM to the register file. `valE` will be written to the register identified by `dstE`, and `valM` will be written to the register identified by `dstM`. In addition, when a halt instruction reaches the writeback stage, the PIPE machine halts the execution of the program.

2.1.3 Pipeline Hazards

When an instruction's result or execution depends upon a previous instruction, a dependence exists between those instructions. When a dependence can potentially cause the wrong result to be produced, it is called a *hazard*. The two types of hazards are data hazards and control hazards.

Data Hazards

A data hazard is a potential erroneous computation due to data dependence. A data dependence occurs when the result calculated by one instruction is needed by another instruction. On the pipeline, the operands are read during the decode stage while a result is written in the later memory and writeback stages. If an instruction causes a register to be modified, that instruction must reach the writeback stage before the result is written to the register file. On the other hand, if an instruction will cause memory to be modified, the instruction reaches the memory stage before the write occurs. In the instructions below, the addq instruction uses the result of the `xorq`. These instructions overlap in execution on the PIPE machine so that when the addq is being decoded, the xorq is being executed. Thus, the value of the `xorq` is not in register `%rax` when the decode is retrieving the operands for the `addq`.

```
xorq %rcx, %rax    #write result in Writeback stage
addq %rsi, %rax    #read operands in Decode stage
```

Control Hazards

A control dependence exists between a conditional jump instruction and the instruction that follows that jump and between the conditional jump and the instruction at the target. Specifically, an instruction A is control-dependent upon another instruction B, if B controls whether or not A will be executed. For example, in the statements below, both

the `addq` and the `xorq` are control dependent upon the `jg` because if the jump is taken, then the `addq` is not executed, and the `xorq` is and vice-versa if the jump isn't taken.

```
            jg target
            addq %rax, %rcx
            ...
target:     xorq %rdx, %rsi
```

Control hazards can occur on the PIPE machine for `ret` and conditional jump instructions because the processor cannot reliably determine the next instruction's location based on the instruction in the fetch stage. On the PIPE machine, when a jump instruction reaches the execute stage, the hardware determines whether the jump is taken or not. Thus, in the cycle in which a conditional jump is fetched, it is unknown whether that jump will be taken. The PIPE hardware handles this by predicting jumps as taken. However, when the jump instruction reaches the execute stage, the processor may realize that the jump should not have taken place. Without repairing the misprediction, erroneous results can occur. This is called a control hazard.

Control hazards can also potentially occur on the PIPE machine when executing `ret` instructions. The return location cannot be reliably determined until `ret` passes through the memory stage, where the processor pops the return address from the stack. The return address is also needed to predict the new PC value for the next instruction. In the PIPE

implementation, return addresses are not predicted. Therefore, processing the `ret` instruction has its control logic, as described later.

Stalling and Bubbling

Stages of the pipeline operate in lockstep. The combinatorial logic of each stage is computed in the first half of the clock cycle. When the clock edge rises, the result produced by the stage is potentially stored in the pipe register that provides input to the next stage. In addition to these inputs, each pipe register is fed control signals known as `normal`, `stall`, and `bubble`. If the normal signal is asserted, then normal behavior occurs (the pipe register stores the value of the input.) This behavior can be seen in Figure 2.4. Each pipeline register has an input, a state, and an output. The input value is the output of the preceding pipe stage. The state of the pipeline register and the output is the current value of the pipeline register. When the clock edge rises, the state of the pipe register either stays the same, becomes the value of the input, or becomes a `nop`, depending upon whether the control signal is `stall`, `normal`, or `bubble`, respectively.

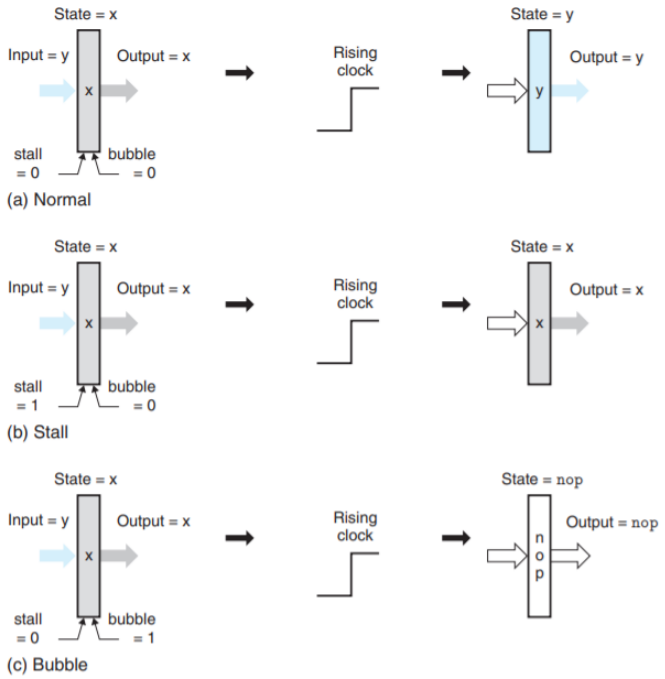

Figure 2.4 Pipeline control logic

In some situations, a pipe register needs to maintain the same value in the next clock cycle that it has in the current cycle to prevent hazards. This occurs when the stall control signal is applied to the pipe register, and the process is known as stalling. If an instruction is stalled in a pipe register, then the stage that register is input to will repeat the same process in the next cycle. For example, stalling the D register will cause the same instruction to be decoded twice.

However, if an instruction is stalled in one pipe register, for example, D, it would be incorrect to apply a normal signal to the next pipe register, E. This would cause the same instruction to be in two pipe registers: D and E. Bubbling is a technique that injects a dynamically generated `nop` instruction into a pipe register. All pipeline registers can

potentially be bubbled, except for the F register. This is because the F register does not contain an instruction; it contains an address.

With stalling and bubbling, the overall flow of the pipeline can be dynamically adjusted to prevent hazards. For data hazards, the PIPE processor can stall the D register until the operands needed by the instruction in the D register are available. To prevent control hazards caused by a `ret` instruction, the pipeline stalls the F register and bubbles the D register for three consecutive cycles until the `ret` instruction reaches the writeback stage.

In addition, bubbling is used to repair branch misprediction by changing the instructions fetched from the target to `nop` instructions. Those instructions are in pipe registers F and D when the conditional branch is being executed, and it is discovered that the branch prediction is wrong. Changing those instructions to `nop` instructions is caused by bubbling the D and E registers. This prevents the instructions in F and D from continuing through the pipeline.

Forwarding

Forwarding is a technique that allows for a pipeline stage to directly send a value to an earlier stage in the pipeline. Values computed by the ALU in the execute stage can be directly sent to the decode stage. Values read from memory and values in the W pipeline register can also be forwarded to earlier stages. For example, in the case of the two instructions below, the `andq` instruction will be decoded in the same cycle that the ALU is calculating the xorq result. The `andq` instruction needs the result of the `xorq`. Rather than

waiting until that the result is written to the register file in the writeback stage, the ALU result will be fed back to the decode stage, where it will be selected by the hardware labeled `Sel+FwdA` in Figure 2.3

```
xorq %rax, %rcx
addq %rcx, %rdx
```

Without forwarding, the hardware would need to stall the `addq` in the D register until the result of the `xorq` is written to the register file.

Control Logic

The control logic of the PIPE machine handles four situations: prevention of load/use hazards, prevention of control hazards due to `ret` instructions, recovering from mispredicted branches, and exceptions. This control logic is expressed in Figure 2.5 using a language known as HCL (hardware control language).

```
# HCL for stalling and bubbling in the Pipeline Register F
bool F_bubble = 0;
bool F_stall =
  # Conditions for a load/use hazard
  E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB } ||
  # Stalling at fetch while ret passes through pipeline
  IRET in { D_icode, E_icode, M_icode };

# HCL for stalling and bubbling in the Pipeline Register D
bool D_stall =
  # Conditions for a load/use hazard
  E_icode in { IMRMOVL, IPOPL } &&
  E_dstM in { d_srcA, d_srcB };

bool D_bubble =
  # Mispredicted branch
  (E_icode == IJXX && !e_Cnd) ||
  # Stalling at fetch while ret passes through pipeline
  # but not condition for a load/use hazard
  !(E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB }) &&
    IRET in { D_icode, E_icode, M_icode };

# HCL for stalling and bubbling in the Pipeline Register E
bool E_stall = 0;
bool E_bubble =
  # Mispredicted branch
  (E_icode == IJXX && !e_Cnd) ||
  # Conditions for a load/use hazard
  E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB};

# HCL for stalling and bubbling in the Pipeline Register M
bool M_stall = 0;
# Start injecting bubbles as soon as exception passes through memory stage
bool M_bubble =
  m_stat in { SADR, SINS, SHLT } || W_stat in { SADR, SINS, SHLT };

# HCL for stalling and bubbling in the Pipeline Register W
bool W_stall = W_stat in { SADR, SINS, SHLT };
bool W_bubble = 0;
```

Figure 2.5 HCL for pipeline control

A load/use hazard can potentially occur when an instruction uses a value read from memory by another instruction. The pipeline will stall the F and D pipeline register and bubble the E register to prevent the hazard. This causes the dependent instruction to remain in the D register until the instruction that reads from memory reaches the memory stage. In HCL, this is expressed as: `E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB }`. Specifically, this expression is true if an instruction in the E register causes a read from memory and if the register destination of that memory instruction matches a source register of an instruction in the decode stage. For example, this can occur with the following set of instructions:

```
mrmovq 0(%rax), %rcx
addq %rcx, %rdx
```

The control logic that handles `ret` instructions is expressed with: `IRET in { D_icode, E_icode, M_icode }`. Specifically, this expression is true if the `icode` in the D register, the E register, or the M register is `ret`. If the expression is true, the F register will be stalled, and the D register will be bubbled. This causes the instruction that sequentially follows the `ret` to be fetched and discarded repeatedly until the `ret` instruction exits the memory stage. The repeated fetch is caused by stalling the F register. Discarding the instruction is caused by bubbling the D register.

Recall that the PIPE machine predicts that jumps are taken by fetching the instruction at the branch target in the cycle after the jump is fetched. Sometimes that

prediction is incorrect. The pipeline will detect the misprediction when the jump instruction reaches the execute stage. The two instructions fetched at the branch target before the misprediction is determined will need to be cleared from the pipeline. Since the branch is in the execute stage when the misprediction is determined, those two instructions are in the decode and fetch stages. To clear them from the pipeline, the D register and the E register are bubbled. The HCL that expresses this logic is `(E_icode == IJXX && !e_Cnd)`. Specifically, this expression is true if the icode in the E register is a jump and the value calculated for `e_Cnd` is 0 (not taken).

When the fetch or execution of an instruction causes an exception, the exception status travels through the pipeline with the instruction. Exceptions are detected during the fetch and memory stages, and the program state is updated during the execute, memory, and writeback stages. When an exception occurs, changes to the program state are disabled for that instruction and the instructions that follow it. When the instruction with an exception reaches the writeback stage, the pipeline halts. Changes to the program state are disabled by bubbling the M register. This logic is expressed by the HCL: `M_bubble = m_stat in { SADR, SINS, SHLT } || W_stat in { SADR, SINS, SHLT }`. Specifically, this indicates that if the instruction with an exception (including a halt) is in the W register or an instruction in the memory stage caused an exception (for example, by a bad memory address), then the M register should be bubbled.

2.2 Angular

Angular is a TypeScript-based open-source framework led by the Angular Team at Google [8]. Most developers use it for building single-page applications, which are web apps that do not require the page to re-render during use. Angular provides much more functionality than what is needed for our simulator. Thus, this section only covers what is necessary to understand the LY86-64 implementation. We used the most updated version of Angular, version 11, which was released on November 11, 2020.

2.2.1 Architecture

Components

Angular applications use a tree of components to define the components' hierarchy. Every component in a typical Angular application consists of an HTML template to display the page's content, a TypeScript class to represent the component's behavior, and a CSS selector. The CSS selector allows us to identify a component in the component tree uniquely. For example, the `ChatComponent` can be represented as the `<chat></chat>` tags in the HTML template by setting the CSS selector as "chat." Additionally, there is an optional CSS file to apply styles to the template.

Angular allows component interactions between a parent component and its children components. A parent component can have zero or more children components, and each parent component can pass information into its children components with input binding. Specifically, when sharing data between parent and children components, data

can flow in two ways: from parent to children and from child to parent. The decorator `@Input` is used to pass data from parent to children. Vice versa, the decorator `@Output` is used to send data from child to parent.

View and source interaction within the component is called data binding. In Angular, the *view* is the HTML template, and the *source* is the TypeScript file. There are three different ways to pass data between the two. Data flows from view to source, source to view, and both ways. Data binding helps with several features like disabling/enabling buttons, displaying computed variables, setting boolean values for CSS, and many more.

Services

Many application elements are dependent on services, which are simple classes with functions and values. Both components and services can be dependent on zero or more services. With components, services are injected into the component with Dependency Injection (DI). Once a service is injected into a component, the component gets access to that service class. There are several reasons why one should use services: to implement logic that is independent of any component, to provide access to shared data and functions, and to allow external interactions like fetching data from a server.

An application-wide mechanism called the *injector* manages a container of reusable dependencies. Besides managing dependencies, the injector also creates them. When a component is dependent on some services, Angular will first check the injector for any reusable instances. If none are found, then the injector will create one for each service and

add them to the container. Once added, the injector returns the services to the component by calling the component's constructor with the services as arguments.

Router

An optional router allows the client-side of the application to navigate between various components. Users can configure a list of route definitions that will translate to `Route` objects. Each `Route` object has two elements: a `path` holding the URL that the user will interact with and a `component` specifying the component to display for this route. When the user interacts with a link, the browser's location changes; if the location change matches a path specified by the router, the router maps to the component matching the route and displays its view.

2.2.2 TypeScript

One of our most prominent reasons for using Angular over other frameworks is that the basis of Angular is TypeScript, a JavaScript superset. TypeScript is an open-source language built by the developers at Microsoft [10]. Like many other languages that perform type-checking, TypeScript allows functions to have return types, declare variables, and write method headers with types.

The TypeScript compiler performs type-checking at compilation time. On the other hand, JavaScript evaluates expressions by treating operands as the same type. For example, the boolean expression `"1" == 1` will evaluate to true in JavaScript due to

implicit checking. In contrast, TypeScript would throw an error. In addition to the many features of JavaScript, TypeScript allows for function overloading while JavaScript does not. It has access modifiers like public, private, and protected for classes to encapsulate their fields and methods. Figure 2.6 provides a summary of TypeScript additions to JavaScript [7].

In addition, TypeScript provides developers the means to use object-oriented programming, where subclasses inherit from superclasses to be used to instantiate objects. On the other hand, JavaScript uses a prototype-based inheritance approach that allows objects to act as prototypes for other objects to inherit from them. Developers more comfortable with object-oriented programming may find using prototype-based inheritance to be awkward.

The TypeScript compiler compiles files with `.ts` extensions into JavaScript files with `.js` extensions. Essentially, this means that all JavaScript code is valid TypeScript code and should compile regardless of type declarations. All of the TypeScript features will then be translated into JavaScript code as well.

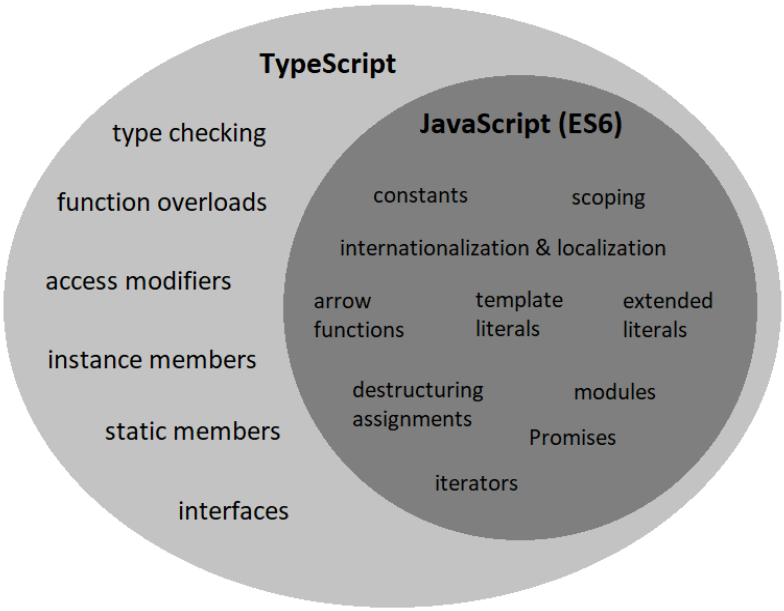

Figure 2.6 TypeScript additions to JavaScript

2.2.3 Summary and Example

In a typical Angular application, many parents and children components will make use of services. The services get injected into each component through their constructors. Once injected, all methods from the injected service are available to the component that depends on it. To navigate between each component on the browser, the developer can configure a URL to each component using the router.

For example, consider a developer-defined component, `HomeComponent`, that has a child component, `ChatComponent`, which needs an array of messages, called `messages`, from the parent. In Figure 2.7, the parent pushes two strings into the `messages` array in its constructor. In the parent's view shown in Figure 2.8, the `HomeComponent` passes `messages` into the `ChatComponent` (represented as `chat` in the view) using the concept

of input binding. The type of binding used here is called “property binding,” which uses the square brackets to set the child component’s `messages` property. The developer then uses `*ngFor`, a built-in Angular template directive equivalent to a `for` loop, to iterate over the `messages` array and print out each element.

```
// home.component.ts -- Parent's source
@Component({
  selector: 'home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css'],
})

export class HomeComponent implements OnInit {
  messages: string[] = [];
  newMessages: string[] = [];

  // injecting an instance of UtilsService into the HomeComponent
  constructor(private utilsService: UtilsService) {
      this.messages.push("Hello, World!");
      this.messages.push("LY86-64");

      this.changeMessages();
  }

  // method that uses the UtilsService
  changeMessages(): void {
    this.newMessages = this.utilsService
                           .addChange(this.messages);
  }
}
```

Figure 2.7: The parent’s source file

```
// home.component.html -- Parent's view
<div class='home'>
  <h1> HOME COMPONENT </h1>
  <chat [messages]="messages"></chat>
  <div class="content" *ngFor="let m of newMessages">
    <p class="messages">{{ m }}</p>
  </div>
</div>
```

Figure 2.8: The parent's view file

Inside the child's source shown in Figure 2.9, the `messages` property has an `@Input` decorator to access the data passed in from the parent. After getting the `messages` array, the child view can now use an `*ngFor` directive to loop through every element and display them. Figure 2.10 is the child's view file.

```
// chat.component.ts -- Child's source
@Component({
  selector: 'chat',
  templateUrl: './chat.component.html',
  styleUrls: ['./chat.component.css'],
})

export class ChatComponent implements OnInit {
  @Input() messages: string[];

  //code goes here
}
```

Figure 2.9: The child's source file

```
// chat.component.html -- Child's view
<div class="chat-div">
  <h1> CHAT COMPONENT </h1>
  <div class="content" *ngFor="let m of messages">
    <p class="messages">{{ m }}</p>
  </div>
</div>
```

Figure 2.10: The child's view file

The developer also defined a `UtilsService` as shown in Figure 2.11 that provides some utilities for the application. In our case, the `UtilsService` has a single method called `addChange()` that takes in a string array as an argument and modifies it by adding "changed" in front of every array element. The `HomeComponent` needs methods from this service, so the developer injected the `UtilsService` into the `HomeComponent` using its constructor. As a result, the `HomeComponent` has access to the `printMessages()` method inside of the `UtilsService`.

```
// utils.service.ts -- Utils service's source
@Injectable({
  providedIn: 'root'
})

export class UtilsService {
  constructor() {}

  addChange(): void {
      let mes = [];
      for (let m of messages) {
            mes.push("changed " + m);
      }
      return mes;
  }
}
```

Figure 2.11: `UtilsService` source file

Afterward, the developer wants to display the `HomeComponent` in the browser, so they assigned a URL to the `HomeComponent` using the router. Figure 2.12 shows an implementation of the router. Now, the clients can see the `HomeComponent` when they visit `/` or `/home` URL in the browser.

```
// app-routing.module.ts -- Router's source
const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: '', redirectTo: '/home', pathMatch: 'full' },
];

@NgModule({
  imports: [RouterModule.forRoot(routes, { relativeLinkResolution:
'legacy' })],
  exports: [RouterModule]
})
export class AppRoutingModule{ }
```

Figure 2.12: The application's router source file

Figure 2.13 demos our working code example. The parent component, `HomeComponent`, has a blue background color, and the child component, ChatComponent, has a red background to help differentiate between the two. The messages inside of the `ChatComponent` are from the `messages` array in the `HomeComponent`. On the other hand, the messages displayed inside the `HomeComponent` are from the `newMessages` array modified by the `changeMessages()` method. When the parent component puts its child component inside its view, the parent component allows the child component to be displayed simultaneously.

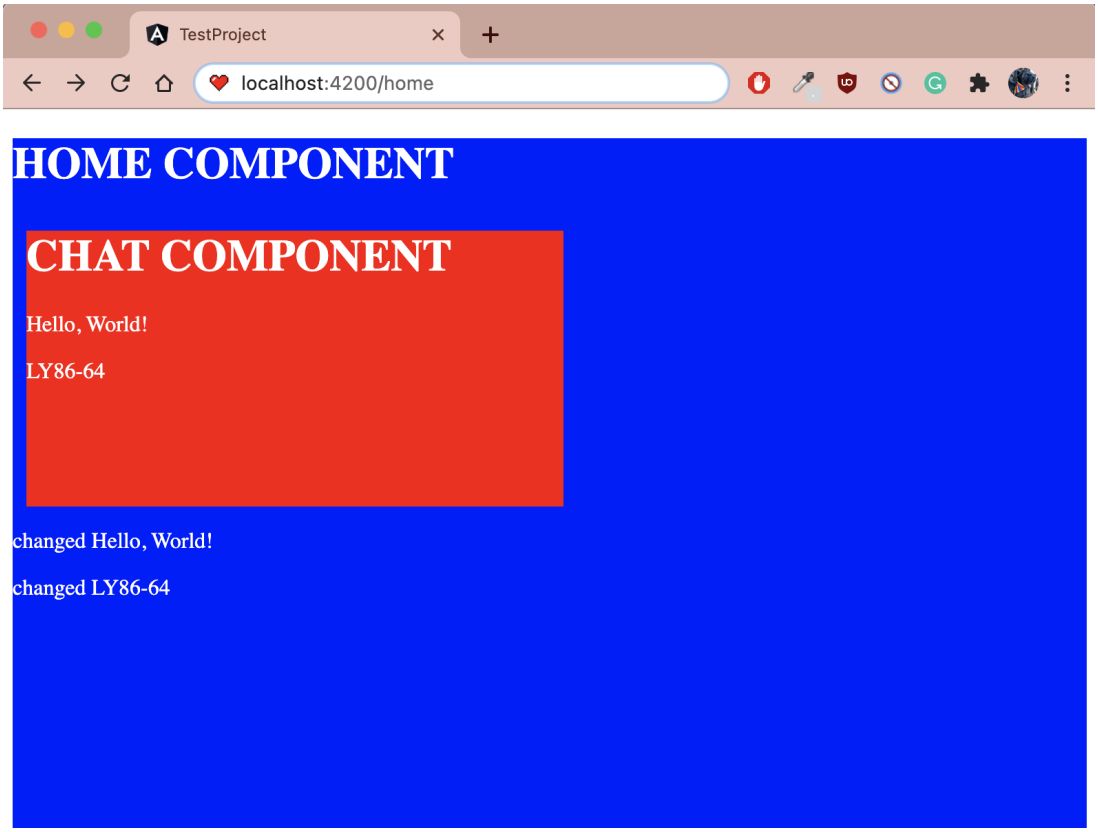

Figure 2.13: Browser display of the HomeComponent and ChatComponent

Chapter 3: Related Work

This chapter provides a description of other Y86 or Y86-64 simulators and emulators and compares them to the LY86-64.

3.1 *YESS: A Y86 Pipelined Processor Simulator*

The YESS (Y-Eighty-Six Simulator) project, developed by faculty at Appalachian State University (AppState) [13], provides a set of labs that can be used to incrementally develop a simulator for the Y86-64 machine. Unlike the other simulators described in this section, YESS was designed to teach students about pipelined machines while implementing a simulator for one. The project provides a set of labs and a set of scripts that students and instructors can use to check for correctness. While the YESS project does not require the student to create a GUI, it does output snapshots of the pipeline registers, register file, and memory for each clock cycle. The YESS project takes `.yo` files as input and generates outputs to the terminal. Figure 3.1 displays a one-cycle snapshot of the output of the YESS project.

```
At end of cycle 12:
F: predPC: 02b
D: stat: 1 icode: 9 ifun: 0 rA: f rB: f valC: 0000000000000000 valP: 02b
E: stat: 1 icode: 6 ifun: 0 valC: 0000000000000000 valA: 0000000000000003
E: valB: 0000000000000001 dstE: 0 dstM: f srcA: 7 srcB: 0
M: stat: 1 icode: 1 Cnd: 0 valE: 0000000000000000 valA: 0000000000000000 dstE: f dstM: f
W: stat: 4 icode: 0 valE: 0000000000000000 valM: 0000000000000000 dstE: f dstM: f

ZF: 0 SF: 0 OF: 0
%rax: 0000000000000004 %rcx: 0000000000000000 %rdx: 0000000000000000 %rbx: 0000000000000000
%rsp: 0000000000000200 %rbp: 0000000000000000 %rsi: 0000000000000000 %rdi: 0000000000000003
% r8: 0000000000000000 % r9: 0000000000000000 %r10: 0000000000000000 %r11: 0000000000000000
%r12: 0000000000000000 %r13: 0000000000000000 %r14: 0000000000000000

000: 000000000200f430 00000003f7300000 00001e8000000000 f030000000000000
020: 0000000000000001 000003f130907060 04f2300000000000 3000000000000000
040: 00000000000003f6 0000000000000000 0000000000000000 0000000000000000
060: 0000000000000000 0000000000000000 0000000000000000 0000000000000000 *
1e0: 0000000000000000 0000000000000000 0000000000000000 000000000000001d
200: 0000000000000000 0000000000000000 0000000000000000 0000000000000000 *
```

Figure 3.1 YESS output

The first line of the output indicates the current cycle of the snapshot. The cycle count is followed by the output of the current values in the pipeline registers F, D, E, M, and W. The next set of output lines are condition flags and register file values. The last set of outputs are the contents of memory.

Similar to YESS, LY86-64 provides a visual representation of the pipeline. However, while the YESS project shows the values of the pipeline registers and memory at each clock cycle, it is difficult to determine from the snapshot when or why the processor stalls or bubbles registers. In addition, YESS was designed for students to build, while the LY86-64 was designed for students to use in order to gain a better understanding of the control logic.

3.2 Bogi Napoleon Wennerstrøm's Y86-64 Simulator

Bogi Napoleon Wennerstrøm's Y86-64 Simulator [15] is an extension of the Y86 Simulator created by Víctor Aguilar The design of the Y86 was based upon the IA-32 instruction set; the Y86-64 is based upon the X86-64. Thus, Wennerstrøm's simulator supports 64-bit signed integer operations as opposed to 32-bit that Aguilar uses. To support 64-bit operations, Wennerstrøm uses the long.js library made by Daniel Wirtz [16]. Importing the long.js library allows access to a `Long` class that represents a 64-bit two's complement integer value.

Wennerstrøm's Y86-64 simulator allows the user the ability to type their own program in the simulator, assemble it, and perform the simulation. As the user steps through each instruction, a green highlight indicates the current instruction being processed. Unlike LY86-64, Wennerstrøm's simulator provides a simulation of a sequential architecture. Specifically, each step displays the change in machine state caused by the execution of one instruction.

Figure 3.2 shows the layout of Wennerstrøm's Y86-64 Simulator. The left component is the code editor in which users can type or paste their Y86-64 assembly program. Clicking the *Assemble* button causes the code to be error checked and assembled. The upper-middle component contains the assembled object code that the simulator uses to perform simulation. The lower middle component displays the registers in the register file, the control flags, the status, and the PC of the simulator. The right component displays memory.

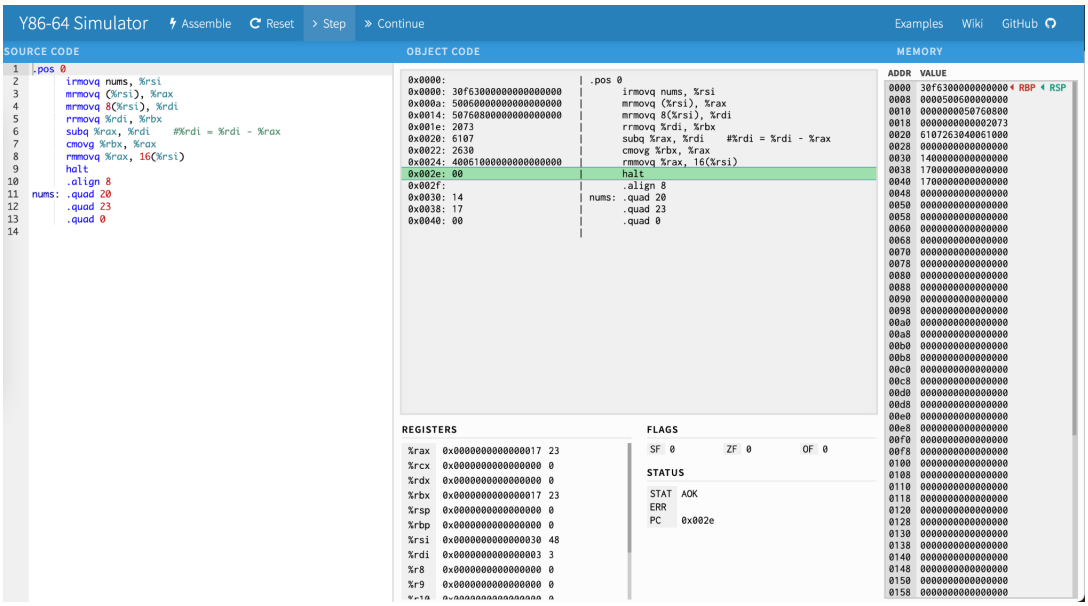

Figure 3.2 Bogi Napoleon Wennerstrøm's Y86-64 Simulator

In the memory component, there are two pointers, RBP and RSP. RBP points to the beginning of the current stack frame, RSP points to the top of the stack. These indicate the current values of the `%rbp` and `%rsp` registers. If the program changes the value of `%rsp` or `%rbp`, these pointers will also change. For example, if the program sets `%rsp` to `0x0050`, then the RSP pointer will move to the `0x0050` location in the memory component.

3.3 Shu Ding's Y86 Emulator

The *Y86 Emulator*, created by Shu Ding [6], is an open-source emulator. This emulator provides a futuristic, sci-fi visualization of the Y86 PIPE machine. Unlike the

Y86-64, whose instructions came from the X86 ISA and supports 64-bit signed integer operations, the Y86 instructions are based on the Intel Architecture, 32-bit (IA-32) ISA.

Ding used AngularJS to build the emulator. While Angular and AngularJS are made by the same team at Google and have very similar names, they are fundamentally two very different frameworks. Angular has a component-based architecture, while AngularJS has a Model-View-Controller (MVC) framework as the central component. It manages data, logic, and controls how the application behaves. Figure 3.3 shows a screenshot of Ding's emulator.

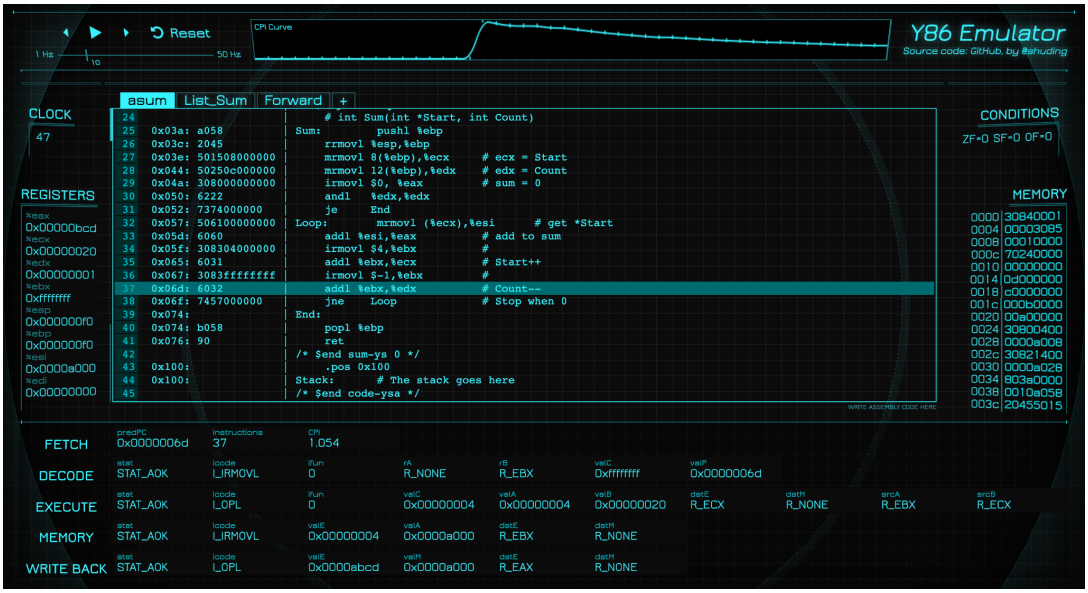

Figure 3.3 Shu Ding's Y86 Emulator

Similar to the YESS project, Ding's emulator also provides a snapshot of the Y86 pipeline at each clock cycle. At the top, there are several buttons to control the flow of the emulation. These buttons provide users the ability to step forward or backward cycle by

cycle, continue execution until the halt is reached, or reset. There is also a cycles per instruction (CPI) counter, a useful feature for performance analysis.

In the center of the emulator is the component that displays the object code. Ding provided three different default files for users to use, or they can upload their own `.yo` file. As the user steps through the instructions, a highlighter is used to indicate the instruction that is about to be fetched. Depending on the instruction, the other components are updated accordingly. Updatable components are registers, condition flags, memory, and pipeline registers. In this emulator, Ding named the pipeline registers "Fetch," "Decode," "Execute," "Memory," and "Write back" to match the pipeline stage's names, rather than F, D, E, M, and W as used in the textbook.

3.4 Linghao Zhang's Y86 Simulator

Linghao Zhang's Y86 Simulator [20] provides a more straightforward visualization than the other simulators described in this chapter and supplies more functionality. In addition to providing a snapshot of the pipeline at each clock cycle, this simulator also simulates a cache, displays performance analysis, and allows the user to save program output into a text file.

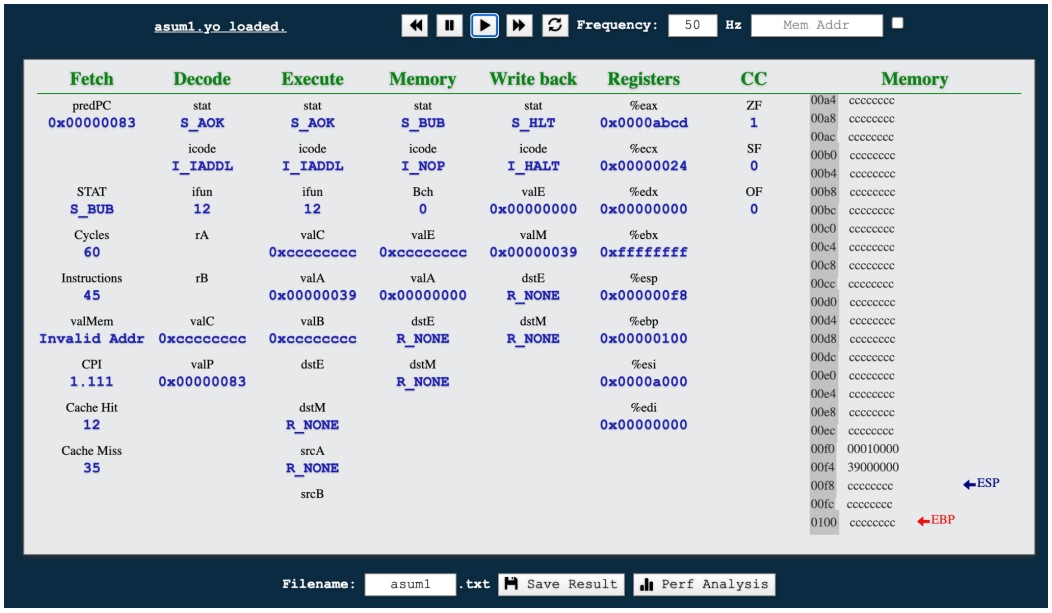

Figure 3.4 Linghao Zhang's Y86 Simulator

Users can simply drag and drop a `.yo` file into the center to upload a file to the simulator. Then, users can use the buttons to control the flow of the simulation. Figure 3.4 shows a screenshot of Zhang's simulator. After running the simulation, users can save outputs into a text file and view the performance analysis.

Figure 3.5 shows the section of the performance analysis that displays cache performance and the performance penalty caused by hardware bubbles. The left pane displays the number of data cache accesses that resulted in a hit or a miss. The right pane displays the frequency of load/use instructions, mispredicted branches, and returns, and the penalties caused by these.

Performance Analysis

Hit/Miss	Count	Percentage
Hit	12	25.5%
Miss	35	74.5%
Total	47	

Cause	InsFreq	CondFreq	Bubble(s)	Product
Load/Use	0.156	0.000	1	0.000
Mispredict	0.133	0.167	2	0.044
Return	0.022	1.000	3	0.067
Total Penalty				0.111

Figure 3.5 Linghao Zhang's simulator performance analysis

3.5 Tianhong Chu's Y86 Simulator

Tianhong Chu's Y86 Simulator [5] is an open-source simulator built using Java. Unlike previously mentioned simulators, Chu's simulator displays the final state of the register file and memory instead of snapshots of each clock cycle. Figure 3.6 provides an example of output for this Y86 simulator. The simulator displays the value of the program counter, the machine status, and the values of the condition codes when the simulation terminates. In addition, the simulator shows the contents of registers and memory before the simulation begins and after the simulation terminates.

```
State: HLT
PC: 0x17
Condition Codes:  ZF: 1 SF: 0 OF: 0
Changed Register State:
%eax:0x00000000     0x0000abcd
%ecx:0x00000000     0x00000024
%edx:0x00000000     0x00000000
%ebx:0x00000000     0xffffffff
%esp:0x00000000     0x00000100
%ebp:0x00000000     0x00000100
%esi:0x00000000     0x0000a000
%edi:0x00000000     0x00000000
Changed Memory State:
0x0000:00000000     0x30f40001
0x0004:00000000     0x000030f5
0x0008:00000000     0x00010000
0x000c:00000000     0x80240000
0x0014:00000000     0x0d000000
0x0018:00000000     0xc0000000
0x001c:00000000     0x000b0000
0x0020:00000000     0x00a00000
0x0024:00000000     0xa05f2045
0x0028:00000000     0x30f00400
0x002c:00000000     0x0000a00f
0x0030:00000000     0x30f21400
0x0034:00000000     0x0000a02f
0x0038:00000000     0x80420000
0x003c:00000000     0x002054b0
0x0040:00000000     0x5f90a05f
```

Figure 3.6 Tianhong Chu's Y86 output

Users are able to view changes to the register file and memory after each instruction by using the `-debug` flag to enter the gdb. In the gdb, the user is able to step through each instruction, run the entire program, break, then run until the next breakpoint, display the instruction at a given address, display a value of a register, or display all changed values in the register file and memory. The Y86 instructions are stepped through instruction by instruction, thus simulating a sequential architecture.

3.6 Comparison

Table 3.1 provides a summary of the differences among the simulators described in this chapter and LY86-64. Several of the works are older and thus are based upon the Y86 ISA, not the Y86-64 ISA. Ding's, Zhang's, and Chu's simulators are all Y86 simulators. Aguilar's simulator, the simulator that Wennerstrøm's simulator extends, is also based on the Y86 ISA. On the other hand, the YESS project, Wennerstrøm's, and LY86-64 are based on the Y86-64 ISA. The YESS project defined a coding project for students to create their own Y86-64 simulator. All others were designed for students to use.

Table 3.1 Comparison Between Other Related Works and the LY86-64

	LY86-64	YESS	Wennerstrøm's	Ding's	Zhang's	Chu's
Y86				✔	✔	✔
Y86-64	✔	✔	✔			
Browser-based	✔		✔	✔	✔	
GUI	✔		✔	✔	✔	
Cache					✔	
Perf Analysis				✔	✔	
Built-in assembler			✔			
Built-in examples	✔			✔		
Control signals	✔					
Pipelined	✔	✔		✔	✔	
Sequential			✔			✔
GDB						✔

Most simulators are browser-based and have a GUI. Chu's simulator and the YESS project do not have a GUI and are not browser-based. Although Chu's simulator does not have a GUI, users can still interact with the simulator using only the gdb. The YESS project, however, does not support user interaction with the running simulator.

Zhang's simulator and Ding's emulator both display the results from performance analysis. Specifically, Ding's emulator displays cycles per instruction. Zhang's simulator outputs the number of misses, hits, and the miss rate assuming a data cache, as well as the penalties caused by hazards. However, the performance analysis usefulness is limited by the size of the input programs as well as the simplistic architecture and ISA. Their results are not likely to reflect the results obtained on modern processors. However, they can help the user understand how those results are calculated.

As mentioned in Section 2.1.2, a sequential architecture performs the execution of a single instruction at a time across the six stages: fetch, decode, execute, memory, writeback, and PC update. Wennerstrøm's and Chu's simulators are both simulating a sequential architecture. They do not display a machine state that reflects instructions overlapping in execution. The *Step* button fetches and executes one instruction to completion, and the output reflects the change in machine state gained by executing that one instruction. There are no pipeline registers in the display.

Several simulators are unique in their features. Chu's simulator is the only one that utilizes gdb. Wennerstrøm's is the only one that allows users to provide Y86-64 assembly code that can then be assembled in the browser. Zhang's simulator is the only implementation that also simulates a cache. Our LY86-64 simulator is the only one that focuses on the control signals.

Chapter 4: LY86-64

In this chapter, we discuss our simulator, LY86-64 simulator, in detail. The source code for LY86-64's implementation can be found at https://github.com/lycb/ly86-64. Section 4.1 provides an overview of the simulator. Section 4.2 explores the design choices, including the layout and color scheme. Section 4.3 discusses the implementation for each component of the simulator.

4.1 Overview

The LY86-64 ("lee 86-64") simulator is a browser-based simulator developed to help students visualize the Y86-64 machine. LY86-64 supports 64-bit signed integer operations. Similar to Wennerstrøm's Y86-64 Simulator, LY86-64 uses the long.js library [16] to gain access to the `Long` class. Although inspired by other Y86-64 simulators, LY86-64 was specifically designed to provide a visualization of the control logic and signals, in particular, stalling and bubbling. Due to the nature of browser-based simulators, users can run the simulation entirely in the browser without the need to download or install any other software.

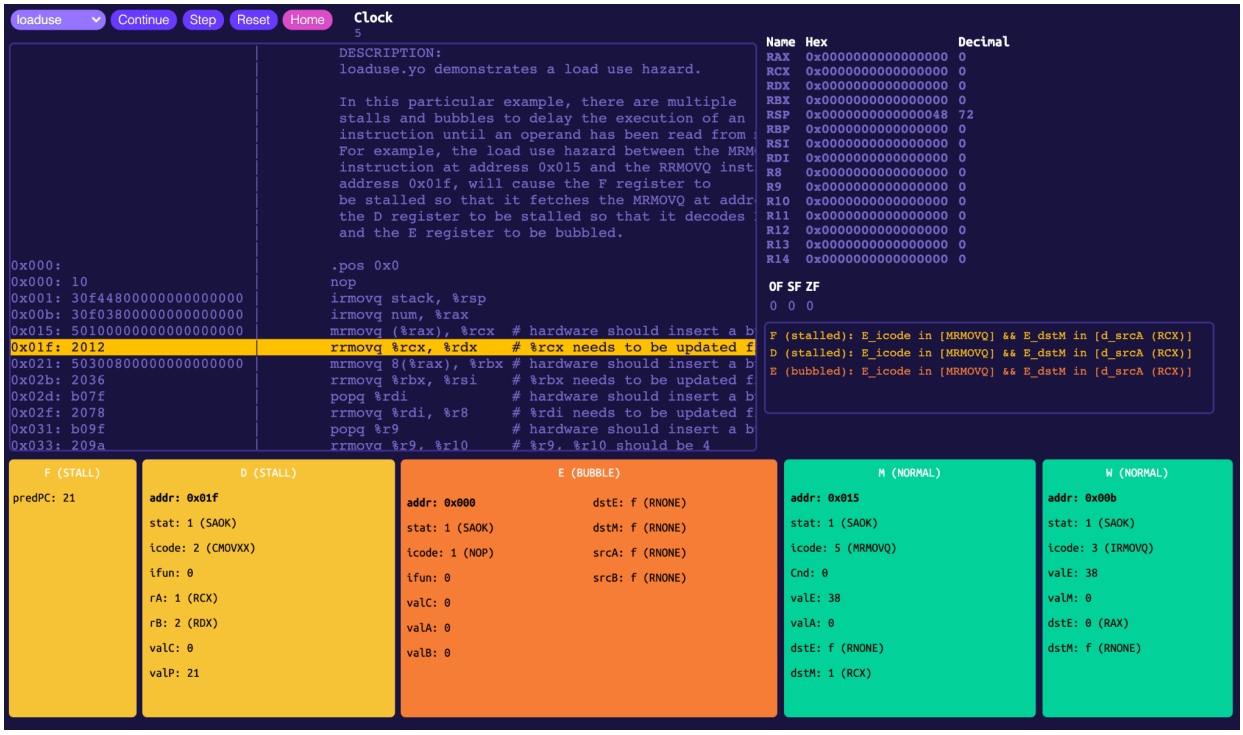

Figure 4.1 LY86-64 simulating a load/use hazard

Figure 4.1 is a screenshot of LY86-64 simulating a load/use hazard. The Y86-64 machine handles the hazard by stalling the F and D pipeline registers and bubbling the E pipeline register. Note that the instruction highlighted at address `0x01f` is using register `%rcx`. When the instruction first reaches the decode stage, `%rcx` does not have the most updated value because the `mrmovq` instruction has not reached the memory stage in order to read the value from memory. LY86-64 provides a visual representation of how the Y86-64 machine handles the hazards by taking advantage of colors. Specifically, the F and D registers are colored yellow to indicate they have been stalled, and the E register is colored orange to show it has been bubbled. In addition, the control logic that determined the

control signal to apply to these registers also appears in the visualization and is similarly colored.

4.2 Design

The LY86-64 display is divided into three separate panes: left, right, and bottom. As their names suggest, these panes are named after their positions on the device's browser. Figure 4.2 shows a wireframe for the LY86-64 website. The left pane has a parent component called "control" that includes a group of control buttons, a clock cycle, and a display for the object code taken from a preloaded or user-uploaded `.yo` file. The right pane contains registers from the register file, condition codes (CC in Figure 4.2), and a component for control logic explanations. Although the left and right panes contain multiple components within themselves, the bottom pane only includes the pipeline register component.

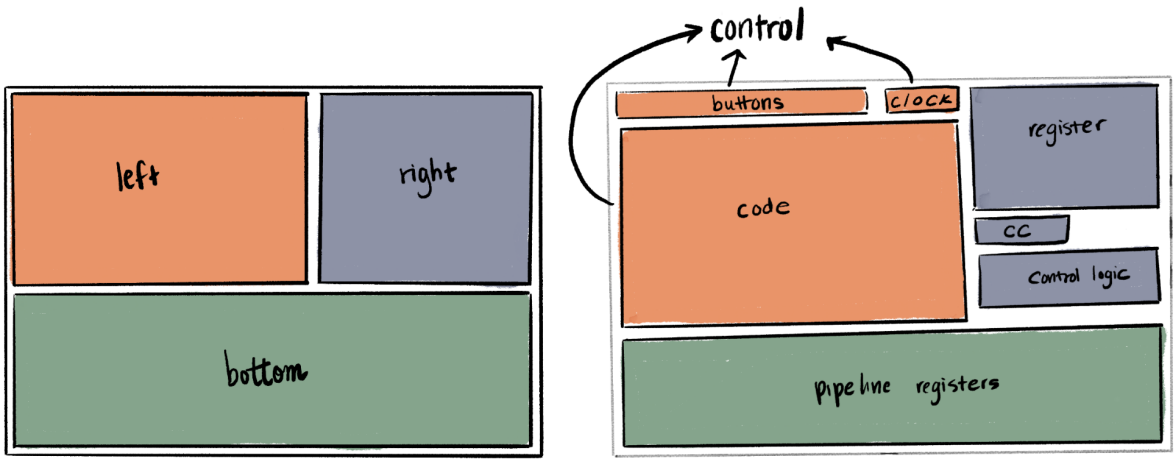

Figure 4.2 Layout wireframe

The left pane contains the buttons that the user will interact with to control the simulation. First, the user must use the dropdown menu to select a pre-loaded `.yo` file or upload their own file. The file is then loaded into the code component. The code component displays the entire file line by line and highlights the first line of code that contains an address. As the user steps through the instructions, the line highlighted is the instruction that was just fetched which is therefore in the D pipeline register.

LY86-64 uses a central service to perform the hardware calculations to mimic the Y86-64 PIPE machine. The right and bottom panes of LY86-64 contain components that resulted from the interaction with the left component. In other words, the left pane supplies the information to be calculated by the central service, and the results are displayed in the right and bottom panes. For each clock cycle, the central service sends the computed data to the components in the right and bottom panes.

Each component that needs data from the central service must inject the service into its own component to access data fields and methods from that service. Recall in Section 2.2.1, the act of injecting a service into a component is called Dependency Injection (DI). Components that inject this central service are the control, register, condition codes, control logic, and pipeline register components. The code and clock cycle components do not need to inject this central service because their parent component, control, directly interacts with the central service. The parent component can pass information to them as they do not need to be involved with any calculations directly.

4.3 Implementation

The LY86-64 simulator has a total of nine Angular components and six services. Table 4.1 lists the services and the hierarchy of the components.

Table 4.1 LY86-64 Component Tree and List of Services

Components			Services
SimulatorComponent	ControlComponent	ClockCycleComponent	ParserService
		CodeComponent	CpuService
	RegistersComponent		RegisterService
	ConditionFlagsComponent		ConditionCodesService
	ControlLogicComponent		MemoryService
	PipelineRegComponent		UtilsService
HomeComponent			

At the highest level of the component tree are the `SimulatorComponent` (LY86-64 simulator) and `HomeComponent` (a homepage for introduction and directions on using the simulator). The purpose for splitting the simulator and the homepage into two separate components is because of Angular routers. The route `/` or `/home` redirects the user to the `HomeComponent`. On the other hand, the `/simulator` leads the user to the `SimulatorComponent` that displays LY86-64. Figure 4.3 is the code representation of the Angular router that our application uses to redirect users.

```
const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'simulator', component: SimulatorComponent },
  { path: '', redirectTo: '/home', pathMatch: 'full' },
];

@NgModule({
  imports: [RouterModule.forRoot(routes,
    { relativeLinkResolution: 'legacy' })],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Figure 4.3 Angular router for the LY86-64 simulator

Within the `SimulatorComponent`, there are five components: `ControlComponent` (main component for controlling the flow of the simulation), `RegistersComponent` (for displaying the register file), `ConditionFlagsComponent` (for displaying the condition flags), `ControlLogicComponent` (for explaining control logic when a hazard occurs), and `PipelineRegComponent` (for displaying the F, D, E, M, and W pipeline registers). Figure 4.4 shows the view file for the `SimulatorComponent`. Recall in Section 2.2.1 that a view file is an HTML template for an Angular component. Each pane, left, right, and bottom, gets their own `<div></div>` for styling and organization purposes. The CSS selector for each component allows the view to recognize a component. For example, the `ControlComponent` has the tag `<app-control></app-control>` and the `RegistersComponent` has the tag `<app-registers></app-registers>`.

```
<div class="container">
  <div class="left">
    <app-control></app-control>
  </div>
  <div class="right">
    <app-registers></app-registers>
    <app-condition-flags></app-condition-flags>
    <app-control-logic></app-control-logic>
  </div>
  <div class="bottom">
    <app-pipeline-reg></app-pipeline-reg>
  </div>
</div>
```

Figure 4.4 View file for `SimulatorComponent`

The LY86-64 simulator also utilizes six services to run the simulation. An essential service is the `CpuService`, as it takes an instruction from the `ControlComponent`, performs calculations for the different pipeline stages, and sends the computed data to other components. However, the `CpuService` depends on several other services to work. Other services include `ParserService` (for parsing instructions into JSON objects), `ConditionCodesService`, `RegisterService`, `MemoryService`, and `UtilsService` (for utility methods). Section 4.3.2 provides more details about these services.

4.3.1 Components

LY86-64 uses Angular components for its GUI. Except for the `ControlComponent`, all other components are for displaying data computed by the `CpuService` and don't include any underlying logic. As discussed in Section 4.2, the components in the left pane (`ControlComponent`) supplies information to a central service (`CpuService`), and the

central service will then send computed information to the components in the right and bottom panes for display. In this section, we discuss the components of LY86-64 with a focus on the `ControlComponent`.

ControlComponent

The `ControlComponent` is a parent component for the `ClockCycleComponent` and the `CodeComponent`. Within the `ControlComponent` are the control buttons that allow users to control the flow of the simulation, a dropdown menu for file selection, and a home button that redirects the user back to the `HomeComponent`. Figure 4.5 is a representation of the `ControlComponent` in HTML. For the dropdown menu, LY86-64 preloaded several files. The files preloaded illustrate various control hazards to help students understand how the Y86-64 machine reacts to such hazards. Specifically, the `jl` example shows how the Y86-64 handles a mispredicted branch, `loaduse` illustrates a load/use hazard, and `addOne` demonstrates control hazards caused by `ret` instructions.

```
<div class=top-button-wrapper>
    <div class="button-wrapper">
        <select id="dropdown" class="custom-select" (change)="onLoadSamples()">
                <option value="choose" selected="selected">Choose</option>
                <option value="upload">Custom files</option>
            <option value="jl.txt" >jl</option>
            <option value="loaduse1.txt" >loaduse</option>
            <option value="addOne.txt" >addOne</option>
        </select>
        <div  class="upload-btn-wrapper" *ngIf="showSelectFile">
        <button class="upload-btn">
            {{uploadButtonText}}
        </button>
        <input  class="file-upload-input" type="file" id="file-input" (change)="onFileSelect()" />
    </div>
        <button class="control-buttons" (click)="onClickContinue()" [disabled]="!loadComponent">Continue
        </button>
        <button class="control-buttons" (click)="onClickStep()" [disabled]="!loadComponent">Step</button>
        <button class="control-buttons" (click)="onClickReset()" [disabled]="!loadComponent">Reset</
        button>
    <button [routerLink]="'/home'" class="regular-buttons" (click)="onClickReset()">Home</button>
    </div>
    <app-clock-cycle [cycle]="cycle"></app-clock-cycle>
</div>

<app-code [fileContent]="fileContent" [dstall]="dstall" [dbubble]="dbubble"
[reset]="reset"></app-code>
```

Figure 4.5 View file for `ControlComponent`

LY86-64 also has the option for users to upload their own `.yo` files. Currently, LY86-64 only supports `.yo` extensions as it cannot parse and assemble `.ys` files. Users must assemble their `.ys` code beforehand [3] or use the pre-loaded files to use the simulator. LY86-64 performs a check for a correct extension to prevent errors and uses a `FileReader` object to read the file's contents asynchronously. The `FileReader` reads the file and separates instruction lines from comment lines, and pushes both types of lines into the `fileContent` array as a `Line` object. The Line object has the following properties: `id`, `textLine` (string representation of the entire line being read), `isAnAddress` (boolean to distinguish between instructions and comments), `isCurrent` (for highlighting the current instruction in the D register), and `parsedLine` (provides properties for an address and the

hex representation of the instruction). The `ParserService` parses the `textLine` and returns an `AddressLine` object with the `address` and `instruction` properties.

With the `fileContent` array populated, the `ControlComponent` passes the array to the `CodeComponent` through input binding. Thus, the `CodeComponent` now has access to the `fileContent` array and can display its contents with CSS styling added to it. Figure 4.6 provides the view file for the `CodeComponent`. There are several CSS classes to distinguish properties for different control logic. For example, a stalled line will have the `class="current-stalled"` attribute, and a bubbled line will have the `class="current-bubbled"` attribute.

```
<div class="file-content-wrapper">
  <div class="file-content" *ngFor="let line of fileContent">
    <pre *ngIf="!line.isCurrent" class="not-current">{{ line.textLine }}</pre>
    <pre *ngIf="line.isCurrent && ((!dstall && !dbubble) || reset)" class="current-normal">{{ line.textLine }}</pre>
    <pre *ngIf="line.isCurrent && dstall && !reset" class="current-stalled">{{ line.textLine }}</pre>
    <pre *ngIf="line.isCurrent && dbubble && !reset" class="current-bubbled">{{ line.textLine }}</pre>
  </div>
</div>
```

Figure 4.6 View file for the `CodeComponent`

The line will be colored a specific color to show the control logic depending on the class attribute. Specifically, suppose a normal signal was applied to the D register. In that case, the just-fetched instruction will be colored purple, an instruction is colored yellow if the D register was stalled, and an instruction will be colored orange if the D register is bubbled. Figure 4.7 shows a rendering of the `CodeComponent`.

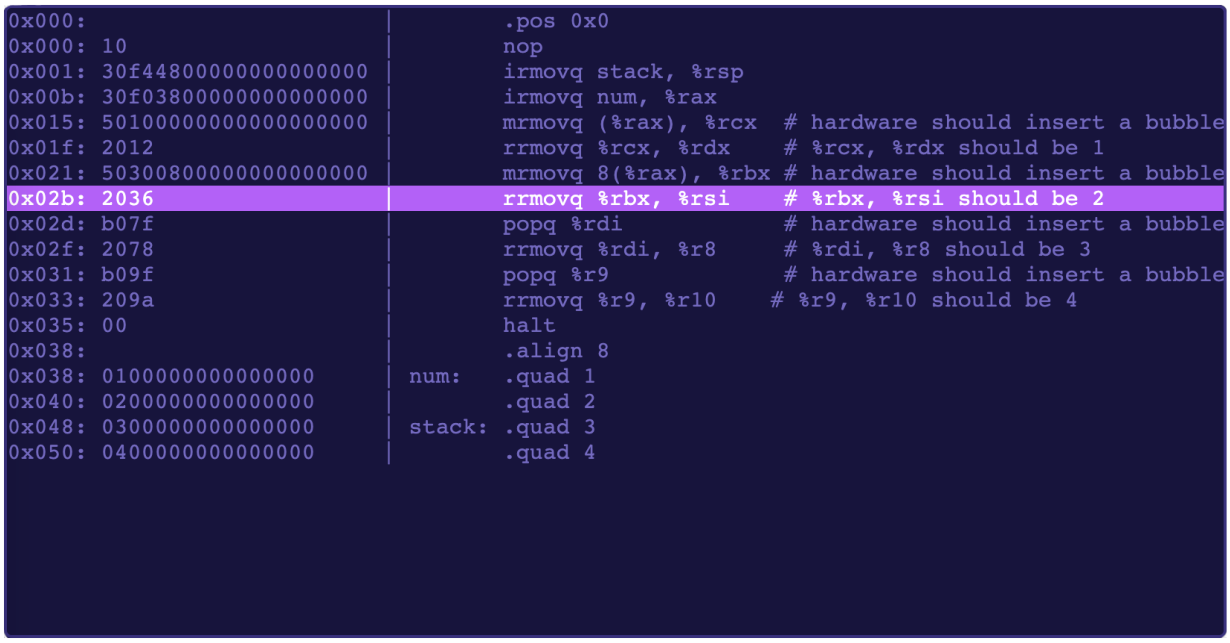

Figure 4.7 Render of the `CodeComponent`

The `ControlComponent` also includes a set of buttons with which the user can control the simulation. The control buttons allow the user to either run through the whole program (*Continue*), step through one instruction at a time (*Step*) or reset the entire program (*Reset*). Each time the user clicks *Step,* the line highlighter highlights the next predicted instruction. Figure 4.8 is a rendering of the control buttons. These buttons are, by default, disabled until the user loads a file into the simulator.

Figure 4.8 Render of the control buttons

Every time the user clicks *Step*, the `onClickStep()` event handler executes. This method sends the current instruction to the `CpuService` and sets the next predicted instruction as the new current instruction. Figure 4.9 is an implementation of the *Step* button event handler. The *Continue* button event handler is a loop of calls to the *Step* button event handler. When the program reaches a halt, a boolean variable, `stop`, is set to true, which prevents the loop from sending any other instructions to the `CpuService`.

```
onClickStep(): void {
  this.reset = false;
  this.isFirstAddressCurrent = false;
  var current = this.parserService.getCurrentLine();
  if (current.parsedLine.instruction == "") {
    this.setFirstCurrent();
    current = this.parserService.getCurrentLine();
  }
  var nextId = current.id + 1;
  if (current.id < this.fileContent.length && nextId < this.fileContent.length) {
    if (current.parsedLine.instruction != "" && !this.stop) {
      this.stop = this.cpuService.doSimulation(this.fileContent, current, this.freg,
                                               this.dreg, this.ereg, this.mreg, this.wreg);
      this.dstall = this.cpuService.getDstall();
      this.dbubble = this.cpuService.getDbubble();
    }
    this.nextCurrentLine();
  }
}
```

Figure 4.9 Step button event handler

`PipelineRegComponent`

The `PipelineRegComponent` in the bottom pane illustrates what the Y86-64 machine does when it encounters a hazard. In addition to using colors to show the registers stalled or bubbled, LY86-64 also informs the user of the control logic by putting the status next to the registers' names. For example, in Figure 4.10, the pipeline register

names are F (STALL), D (STALL), E (BUBBLE), M (NORMAL), and W (NORMAL) to indicate that the F and D registers were stalled, the E register was bubbled, and the normal signal was applied to the M and W registers. LY86-64 displays all values for each pipeline register. Some registers also include an address (`addr`) to make it easier for users to connect the contents of the pipeline register to the specific instruction in the Y86-64 input program.

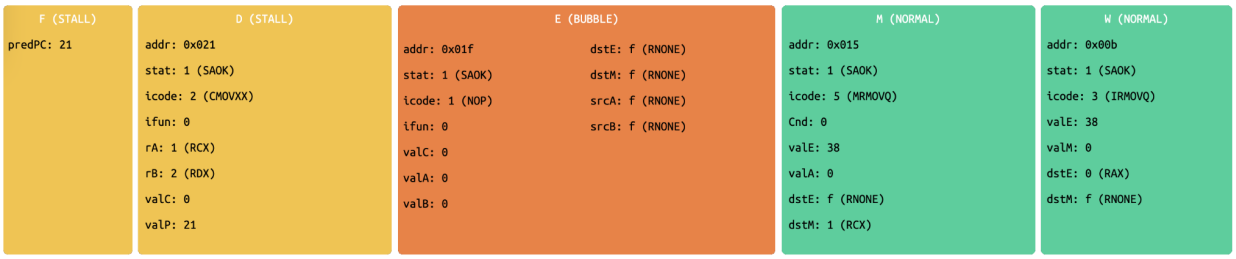

Figure 4.10 Render of the `PipelineRegComponent`

Other Components

The `RegistersComponent`, `ConditionFlagsComponent`, and `ControlLogicComponent` display values calculated by the `CpuService`. The `RegistersComponent` shows the fifteen registers in the register file. The LY86-64 provides a 64-bit hex representation and a decimal representation of the value in a register. The decimal representation is convenient since the immediate values in the Y86-64 code may be written in decimal. The `ConditionFlagsComponent` displays the condition codes: OF, SF, and ZF. Each of these condition codes displays either a 0 or a 1 under their respective labels to indicate whether the execution of the most recent `OPq` instruction produced an overflow, a negative result, or a zero. The `CpuService` sends a value every clock cycle to

allow the `RegistersComponent` and the `ConditionFlagsComponent` to be updated, even if the values are the same as the previous clock cycle.

While the `PipelineRegComponent` illustrates what happens when the Y86-64 machine handles a control hazard, the `ControlLogicComponent` displays the HCL to explain why a certain pipeline register stalled or bubbled. The strings for each pipeline register are formed within the `CpuService` by doing simple string concatenation. For each clock cycle, the `CpuService` sends the string to the `ControlLogicComponent` for display. Figure 4.11 is a render of the `ControlLogicComponent` during a load/use hazard.

```
F (stalled): E_icode in [MRMOVQ] && E_dstM in [d_srcA (RCX)]
D (stalled): E_icode in [MRMOVQ] && E_dstM in [d_srcA (RCX)]
E (bubbled): E_icode in [MRMOVQ] && E_dstM in [d_srcA (RCX)]
```

Figure 4.11 Render of the `ControlLogicComponent`

4.3.2 Services

Unlike components, services are not visible to the user. Multiple components can use the same service, and services do not have a hierarchy tree to dictate data flow. Components use services by injecting them into the components' source files. Services implement logic independent of any component and provide access to shared data and functions to any component a service is injected into.

ParserService

When the user first loads a file into LY86-64, the `ControlComponent` uses the `ParserService` to parse instructions. Every instruction has an address and a hex representation of that instruction. For example, the line `0x001:` `30f44800000000000000` has the address 1 (`0x001`) and the instruction `30f44800000000000000`. As mentioned in Section 4.3.1, the `ParserService` parses the `textLine` (the line read from the input file) into an `AddressLine` object with the properties `address` and `instruction`. LY86-64 uses a regular expression to parse the `textLine`. To prevent the `ParserService` from accidentally parsing non-instruction lines (comment lines), LY86-64 performs a check to see if there are both an address and instruction bytes in the `textLine` before parsing.

CpuService

The main entry to using the `CpuService` is from the `ControlComponent`. The `onClickStep()` event handler for the *Step* button calls the `doSimulation()` method within the `CpuService`. This method performs the simulation of each pipeline stage (fetch, decode, execute, memory, and writeback), sets the control logic strings, and tells the LY86-64 whether to stop the simulation. Figure 4.12 provides the implementation of the `doSimulation()` method.

```
doSimulation(fileContent: Line[], lineObject: Line, freg: F, dreg: D,
                    ereg: E, mreg: M, wreg: W): boolean {
  this.logic_string = ["", "", ""];

  let stop = this.doWritebackClockLow(wreg);
  this.doMemoryClockLow(lineObject, freg, dreg, ereg, mreg, wreg);
  this.doExecuteClockLow(lineObject, freg, dreg, ereg, mreg, wreg);
  this.doDecodeClockLow(lineObject, freg, dreg, ereg, mreg, wreg);
  this.doFetchClockLow(fileContent, lineObject, freg, dreg, ereg, mreg, wreg);

  if (this.fstall) {
    this.logic_string[0] = this.f_logic_string;
  }
  if (this.dstall || this.dbubble) {
    this.logic_string[1] = this.d_logic_string;
  }
  if (this.ebubble) {
    this.logic_string[2] = this.e_logic_string;
  }
  if (this.mbubble) {
    this.logic_string[3] = this.m_logic_string;
  }

  this.doWritebackClockHigh(wreg);
  this.doMemoryClockHigh(wreg);
  this.doExecuteClockHigh(mreg);
  this.doDecodeClockHigh(ereg);
  this.doFetchClockHigh(freg, dreg);
  this.logic.next(this.logic_string);

  return stop;
}
```

Figure 4.12 Simulation logic inside of the `CpuService`

Recall in Section 2.1.3 that each pipeline register has an input, a state, and an output. The input value is the output of the preceding pipe stage. The state of the pipeline register and the output is the current value of the pipeline register. For each clock cycle, the clock starts as low, then rises to a high. LY86-64 simulates these behaviors of clock lows and clock highs with two sets of functions. For each stage, the `CpuService` has clock low and clock high methods. For example, the fetch stage has `doFetchClockLow()` and `doFetchClockHigh()` methods. The clock low method simulates the combinational logic within the stage. The clock high method updates the state of the pipeline register. Depending on the control signal, the state either stays the same (stall), becomes the value

of the input (normal), or becomes a `nop` (bubble). The `CpuService` also calls these clock low and high methods in reverse order to mimic the parallel behavior of the hardware. The order is as follows: write back, memory, execute, decode, then fetch.

Miscellaneous Services

`ConditionCodesService`, `RegisterService`, and `MemoryService` provide functions to help manage the condition codes, register file, and memory aspects of the simulator. The `UtilsService`, on the other hand, has utility functions like translating a register's name to its respective numeric representation, translating `icode` and `ifun` to instructions, padding binary and hex, checking for overflow, and many more.

`ConditionCodesService` provides getters and setters to allow `CpuService` access to the condition codes. During the execute stage, `CpuService` builds an array of condition codes to send to `ConditionFlagsComponent` for display.

The constructor of the `RegisterService` creates an array of objects, each representing a register in the register file. The default object, before changes are made to the register file, contains the name of a register, a `Long` object for the number zero, and a hex string for `0x0000000000000000`. `RegisterService` also provides getter and setter methods that allow `CpuService` to set and get values by the name of the register. This is supported by using the `register2index()` method from `UtilsService` for translating the register's name to its numeric representation. For example, the index 0 in the array of registers from `RegisterService` can be set by calling `setValueByRegister("RAX",`

`valE)`. This call updates both the hex string and the decimal representation to `valE` for index 0 (register `%rax`).

Similar to `RegisterService`, `MemoryService` also has an array of Long objects to represent memory. Each element contains a byte of data. Every multiple of eight in this array contains a value of size 64 bits. `MemoryService` provides functions to get and set a byte of data or 64 bits of data. Functions to get or set 64 bits of data using a method in the `UtilsService` to build a `Long` object from the bytes in the memory array.

Chapter 5: Results

Appalachian State University offers two systems courses, which both utilize the Bryant and O'Hallaron textbook [4]: Computer Systems 1 and Computer Systems 2. The Computer Systems 1 course covers data representation, machine-level representation of programs, and processor architecture. Students in the Computer Systems 1 course are asked to implement a simulator for the Y86-64 PIPE machine in C++. That project is described in Section 3.1. Among the most difficult concepts for students to understand is the topic of processor controls signals, specifically the need for and the impact of stalling and bubbling pipeline registers. The purpose of the development of LY86-64 was to enable students to better understand those concepts.

After developing LY86-64, we asked students from Appalachian State University who are currently taking Systems 1 or took Systems 2 in the previous semester to use our simulator and complete a survey. In this chapter, we compare and discuss the results of the surveys from the two groups of users: those who were learning about the Y86-64 machine when they completed the survey (Group 1) and those who had studied it in a previous semester (Group 2).

5.1 User Experience

A total of 47 students experimented with the LY86-64 simulator and completed the survey. Thirty-one of those students were in Group 1 (current experience), and 16 of those students were in Group 2 (past experience).

5.1.1 User-friendliness

The decision was made to implement a browser-based simulator over a command-line application or downloadable software in order to allow the use of the simulator without requiring installation and to avoid compatibility issues between different operating systems. We expected that users would work with a variety of different operating systems such as Windows, macOS, and Linux distributions. In addition, users, in particular students, are typically not administrators of the machines available for educational use, and it is troublesome or impossible for users to download software on those machines. With these concerns in mind, a browser-based simulator was determined to be the best option. Any user with a browser can access it without admin restrictions and operating system compatibility issues. From the survey results, we learned that every student could access LY86-64 from their choice of browser. The list of browsers that students used includes Chrome, Firefox, Safari, Internet Explorer, Microsoft Edge, and Brave.

While LY86-64 does not allow users to paste or type their own program, it does allow the user to upload their own `.yo` file. To upload a file, the user must select “Custom files'' in the dropdown menu in LY86-64. In Group 1, 74.2% of participants either strongly

agree or somewhat agree that it is easy to upload their own file, 16.1% were indifferent, and 9.7% did not upload their own file. In Group 2, 56.3% of participants either strongly agree or somewhat agree, and 43.8% did not upload their own file. With these results, we can conclude that most users think uploading a file is easy to do. 0% of participants strongly disagree with the statement, "It is easy to upload my own `.yo` file." Figure 5.1 shows pie charts for users' thoughts on uploading their own `.yo` file.

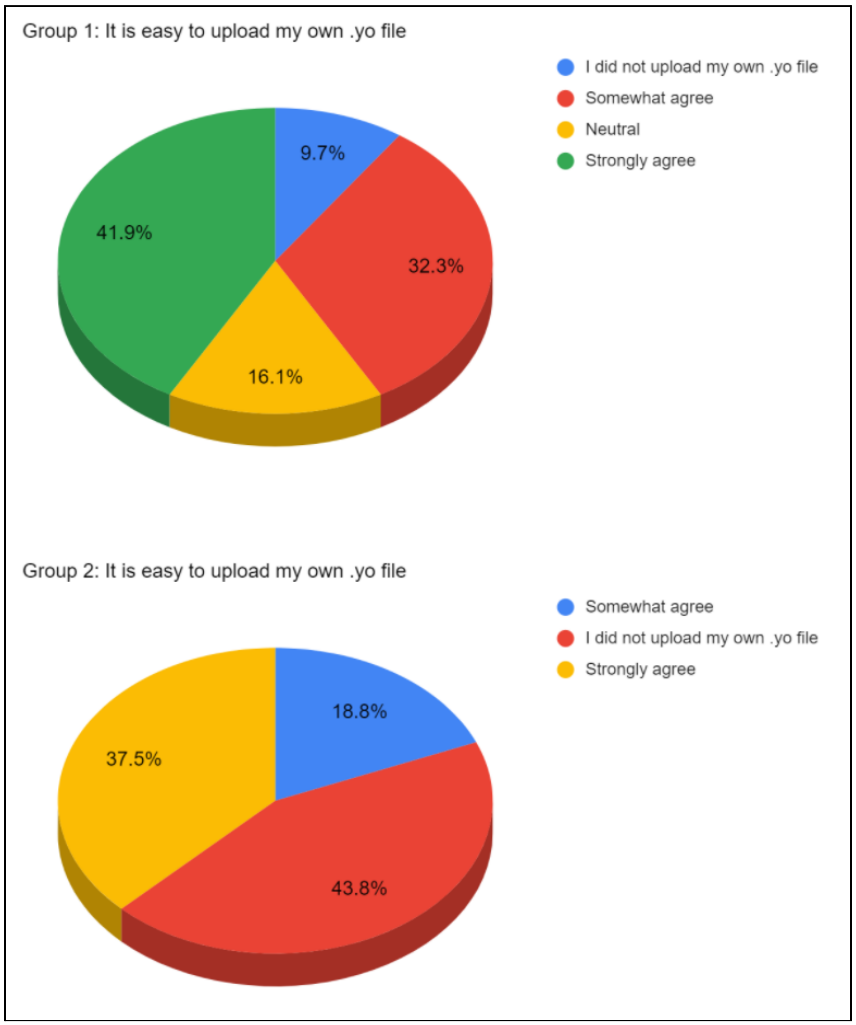

Figure 5.1 User opinion on uploading a `.yo` file

Other survey statements include “Stepping through the instructions is…,” “Resetting the simulator is…” with the majority of participants thinking that both tasks are either very easy or somewhat easy to do. This finding was consistent among both Group 1 and Group 2. Figure 5.2 shows the results for what users think about stepping through the instructions.

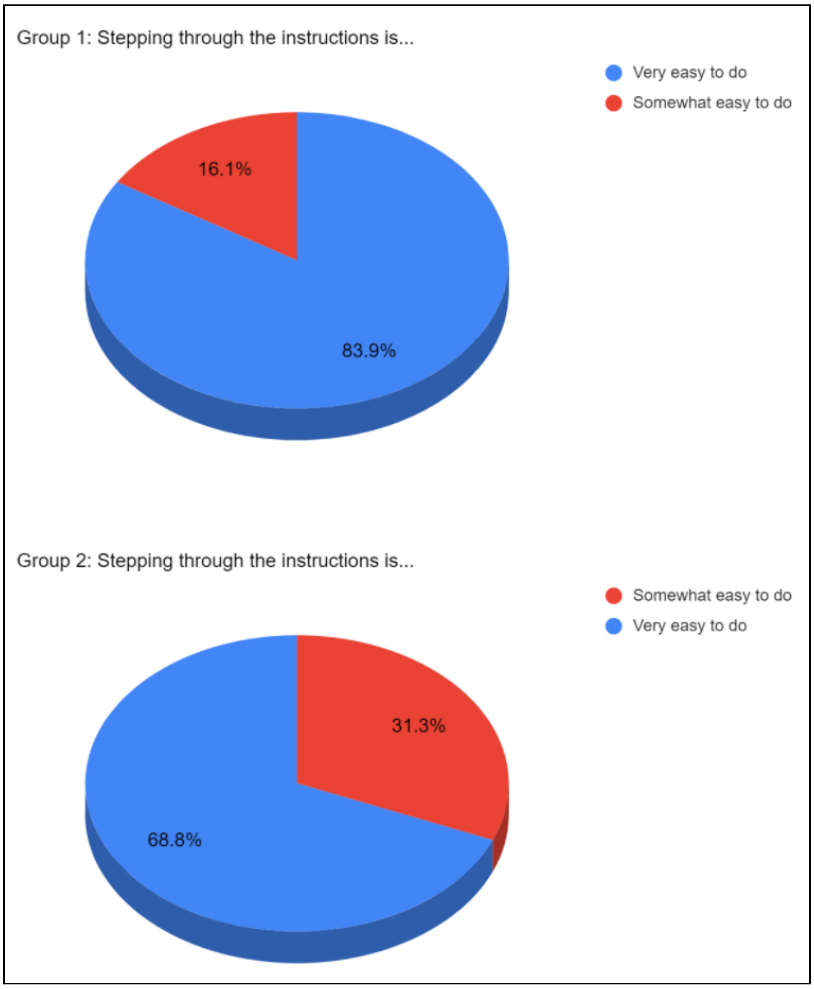

Figure 5.2 User opinion on stepping through instructions

5.1.2 Output Understanding

The `HomeComponent`, as mentioned in Section 4.3, is a component specifically for the homepage that users can access by going to the `/home` or `/` routes. Here, we have a section for the LY86-64 documentation to explain specific colors, functionalities for each component, etc. Among the respondents, 93.5% of users in Group 1 and 87.5% of users in Group 2 read the documentation. With these results, we expected both groups to have a general understanding of the colors in the `PipelineRegComponent` and `CodeComponent`, and the HCL in the `LogicComponent`.

Figure 5.3 shows the responses for Group 1 and Group 2 in regards to their understanding of the colors applied to the pipeline register component. (Recall, a pipe register was colored green if the normal signal was applied to it, yellow if the stall signal was applied, and orange if the bubble signal was applied.) 93.5% of Group 1 understood the use of colors in LY86-64 right away; this is the same percentage of respondents who read the documentation. However, only 68.8% of Group 2 understood the colors immediately even though 87.5% read the documentation. Regardless, with additional experimentation, 100% of users ultimately understood the use of colors in the simulator. This shows that the use of colors in the LY86-64 simulator is easy to understand.

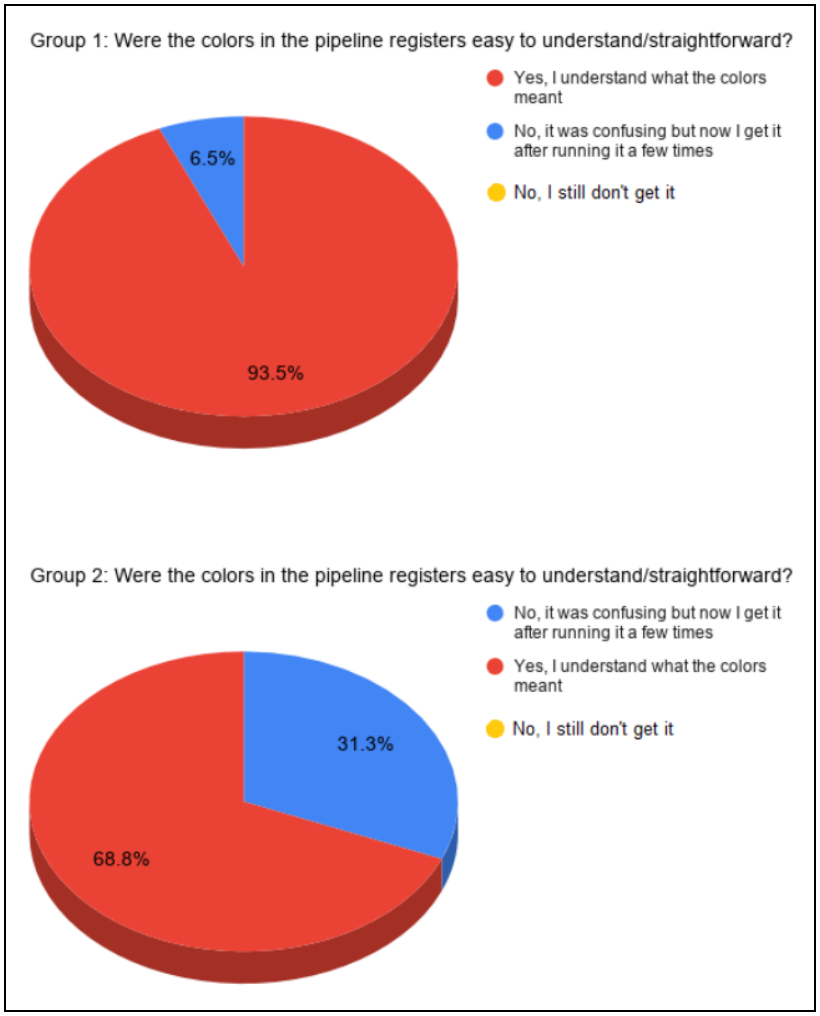

Figure 5.3 User understanding of colors

Recall that the `LogicComponent` displays the HCL that determines why certain pipeline registers are stalled or bubbled. In Group 1, we find that 96.8% of users understood the HCL. While in Group 2, only 75% of users understood. This result indicates that users who are currently learning about the Y86-64 are more likely to understand why the pipeline registers stalled or bubbled. Figure 5.4 contains pie charts that illustrate the users' perceived understanding of the HCL in the `LogicComponent`.

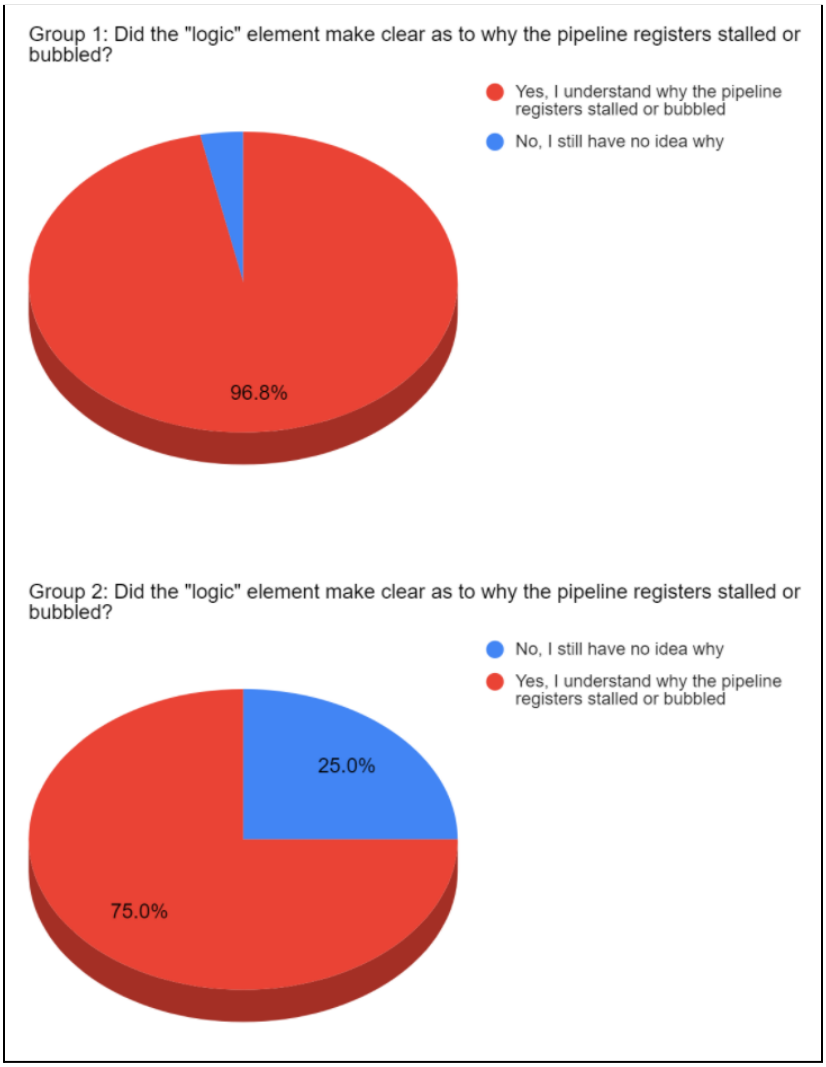

Figure 5.4 User understanding of the HCL

5.2 Improved Understanding

LY86-64 was developed in order to allow students to develop a better understanding of the difficult concepts of stalling and bubbling. Since the students in Group 1 were currently studying those concepts when they experimented with LY86-64, the

survey developed for that group specifically asked them whether the simulator improved their understanding. Figure 5.5 shows a chart displaying the result of that survey question; 96.8% of the students indicated an improved understanding.

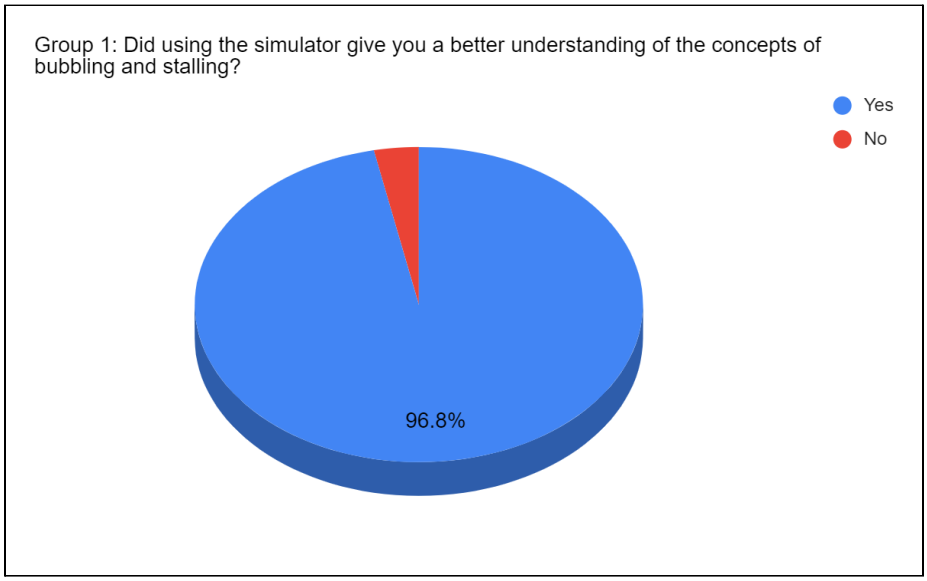

Figure 5.5 Group 1 understanding of stalling and bubbling

Since Group 2 had studied the material in a previous semester, we asked the participants their perceived understanding of the concepts of bubbling and stalling when they were students in CS 3481. Among Group 2 respondents, 12.5% indicated minimal understanding of the concepts of bubbling and stalling, 56.3% understood the material pretty well or completely, and 31% provided a neutral response. Figure 5.6 shows a chart displaying group 2's understanding of the concepts of stalling and bubbling. However, after using the LY86-64 simulator, 100% of participants from Group 2 answered yes to the

question, “Could this simulator enable students to develop a better understanding of the concepts of bubbling and stalling?”.

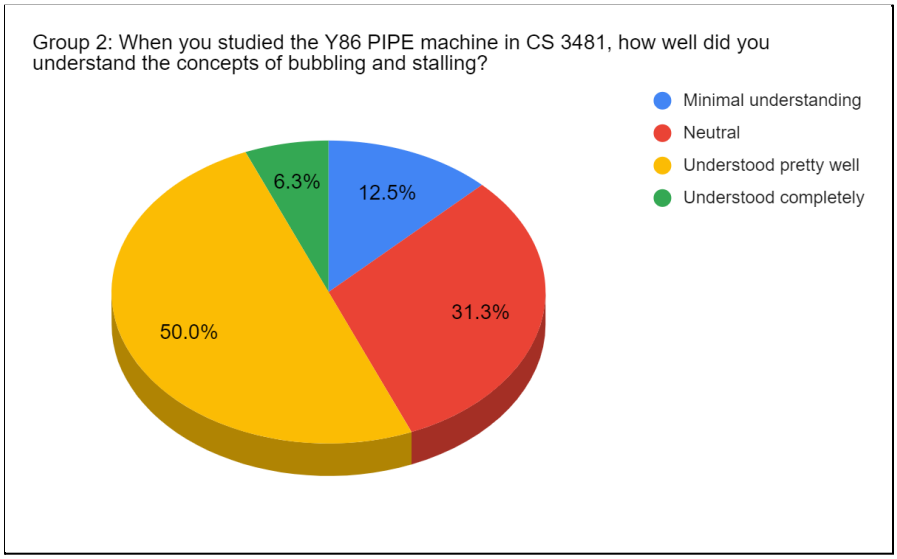

Figure 5.6 Group 2 understanding of stalling and bubbling

Chapter 6: Future Work

This thesis described the design and evaluation of a functional, browser-based GUI that simulates the Y86-64 PIPE machine. LY86-64 allows users to upload a Y86-64 program or select a built-in Y86-64 program and step through the execution of the program cycle by cycle. While stepping through the program, the contents of general purpose registers, pipeline registers, and memory are displayed. In addition, each pipeline register is colored in a way that indicates what control signal (normal, stall, or bubble) was applied to the register. The GUI also displays the Hardware Control Logic (HCL) that determined those signals.

Students who had either taken Computer Systems 1 or were enrolled in Computer Systems 1 were recruited to experiment with the LY86-64 simulator and complete a survey. Survey results indicated that the simulator functioned properly on a broad set of browsers and was easy to use. In addition, nearly 100% of the respondents indicated that the simulator could improve their understanding of the concepts of bubbling and stalling. The remainder of this chapter discusses future work.

Effectiveness Study

Although the surveys given to current and former Systems I students provided promising results, they do not provide the best measurement of improved understanding. Students may believe the simulator increased their understanding of the Y86-64 PIPE

machine control logic when it actually did not. A more robust study of the effectiveness of LY86-64 will be undertaken in Fall 2021. Students taking the Systems 1 course will take a quiz that contains questions about the Y86-64 PIPE machine control logic. After the first quiz, students will be asked to use the LY86-64 simulator. After using the simulator, students will take another to measure a change in the understanding of the concepts of stalling and bubbling.

Error Checking For User-Uploaded Files

LY86-64 currently provides no error checking for user-uploaded files other than checking whether a file has a `.yo` extension before running the simulation. In addition, the `.yo` extension is used both for files containing Y86-64 instructions and for files containing Y86 instructions. If the user uploads a `.yo` file containing Y86 instructions, the simulator will not work correctly. Error checking a file can include checking if the instructions are properly formatted and encoded. For example, a valid encoding for an instruction starts with `0x` and is followed by three hex digits that identify the location in memory in which the instruction will be loaded. The address is followed by a colon and a space, followed by one to ten bytes, in hex, for the instruction encoding. At present, if the user uploads a `.yo` file with errors, unpredictable results can occur. Error checking the input file would be particularly useful for users writing the encoding of instructions by hand.

Accessibility for Those With Visual Impairments

This simulator relies heavily on colors to help users understand the control logic. Unfortunately, if the user has a visual impairment, LY86-64 may exclude such users from being able to learn from the simulator. Visual impairment can include but is not limited to color blindness and blindness. Increasingly, technology is being designed in a way that does not prohibit use by users with visual impairments. For example, the game League of Legends added a colorblind mode in 2012 for players with deuteranopia (red-green colorblindness) to help players distinguish between the friendly's minions and the foe's minions [11]. Currently, the LY86-64 simulator provides some support for those with visual impairments by including labels next to the pipeline registers' headings and in the logic component. For example, F (STALL) indicates the F register is stalling. However, we want users with other visual spectrums to fully experience our simulator's colorful visuals. We also want to accommodate users with a blindness disability that rely on screen readers, refreshable braille displays, and speech recognition software to access technology. This type of technology falls under the category of Assistive Technology (AT), which describes hardware or software that helps people with disabilities to use technology. In the future, we want blind individuals to be able to use LY86-64 with just a keyboard to navigate between the elements.

UI Changes

LY86-64 currently uses a darker purple background and the smallest browser size recommended is 544x807, assuming the user's browser is zoomed in at 100%. To make

LY86-64 mobile and tablet-friendly and provide a dark and light mode, some UI changes are needed. While the size recommended fits most tablet sizes, it does not fit all. Smaller tablets can be supported by adding mobile and tablet-supported CSS properties. However, fitting a layout meant for a bigger screen on a smaller screen like a smartphone can be challenging. This requires redesigning the layout for a smaller screen. Regarding colors, some participants in the survey suggested adding a dark and light mode toggle to the UI to better attract and engage users.

Delay Animation Between Each Cycle

The *Continue* button currently runs through the entire program until the program halts without any delay. In other words, when the user clicks *Continue*, the simulator updates every value to the very last clock cycle. Another enhancement would be to provide a cycle-by-cycle animation of the pipeline that does not require the user to press *Step*.

Forwarding

Like stalling and bubbling, forwarding is another challenging concept for students to grasp. As mentioned in Section 2.1.3, forwarding is a technique that allows for a pipeline stage to directly send a value to an earlier stage in the pipeline. Since there are a number of values from both the pipeline stages and pipeline registers that can be forwarded, adding of visual aids (for example, highlighting values forwarded) would be difficult as the user cannot see values in the pipeline stages. Although LY86-64 simulates forwarding, the `LogicComponent` currently does not display anything to inform the user that forwarding

of values occurred. An improvement would be to illustrate the forwarding logic in the HCL or to redesign the layout altogether to provide a visualization of forwarding.

References

[1] Rashmi Agrawal, Sahan Bandara, Alan Ehret, Mihailo Isakov, Miguel Mark, and Michel A. Kinsy. 2019. The BRISC-V Platform. Proceedings of the Workshop on Computer Architecture Education - WCAE'19 . DOI:http://dx.doi.org/10.1145/3338698.3338891

[2] Miloš Bečvář and Stanislav Kahánek. 2007. VLIW-DLX simulator for educational purposes. *Proceedings of the 2007 workshop on Computer architecture education - WCAE '07* . DOI:http://dx.doi.org/10.1145/1275633.1275636

[3] Randal E. Bryant and David R. O'Hallaron. 2015. APP3e Student Site. Retrieved April 30, 2021 from http://csapp.cs.cmu.edu/3e/students.html

[4] Randal E. Bryant and David R. O'Hallaron. 2016. Computer systems: A Programmer's Perspective 3rd ed., Harlow, United Kingdom: Pearson.

[5] Tianhong Chu. 2018. Y86-Simulator. Retrieved April 29, 2021 from https://github.com/CtheSky/Y86-Simulator

[6] Shu Ding. 2017. Y86 Emulator. Retrieved September 29, 2020 from https://github.com/shuding/y86

[7] Ralf S. Engelschall. 2017. Retrieved April 29, 2021 from http://es6-features.org/

[8] Google. 2021. Angular Documentation. Retrieved April 29, 2021 from https://angular.io/docs

[9] Kenneth E. Hoganson. 2002. High-performance computer architecture and algorithm simulator. Journal on Educational Resources in Computing 2, 1 (2002), 131–148. DOI:http://dx.doi.org/10.1145/545197.545204

[10] Microsoft. 2021. TypeScript Documentation. Retrieved April 29, 2021 from https://www.typescriptlang.org/docs/

[11] Moonopal. 2012. Colorblind Mode. Retrieved April 29, 2021 from https://support-leagueoflegends.riotgames.com/hc/en-us/articles/201752844-Colorblind-Mode

[12] Seikoh Nishita. 2004. MKit simulator for introduction of computer architecture. Proceedings of the 2004 workshop on Computer architecture education held in conjunction

with the 31st International Symposium on Computer Architecture - WCAE '04 (2004). DOI:http://dx.doi.org/10.1145/1275571.1275598

[13] Cindy Norris and James Wilkes. 2007. YESS: a Y86 pipelined processor simulator. Proceedings of the 45th annual southeast regional conference on - ACM-SE 45. DOI:http://dx.doi.org/10.1145/1233341.1233369

[14] Kian L. Pokorny. 2015. Creating a Computer Simulator as a CS1 Student Project. Proceedings of the 46th ACM Technical Symposium on Computer Science Education (2015). DOI:http://dx.doi.org/10.1145/2676723.2677210

[15] Bogi Napoleon Wennerstrøm. 2017. js-y86-64. Retrieved April 29, 2021 from https://github.com/boginw/js-y86-64

[16] Daniel Wirtz. 2018. long.js. Retrieved April 29, 2021 from https://github.com/dcodeIO/long.js

[17] Gregory S. Wolffe, William Yurcik, Hugh Osborne, and Mark A. Holliday. 2002. Teaching computer organization/architecture with limited resources using simulators. Proceedings of the 33rd SIGCSE technical symposium on Computer science education - SIGCSE '02 (2002). DOI:http://dx.doi.org/10.1145/563340.563408

[18] Cecile Yehezkel, William Yurcik, Murray Pearson, and Dean Armstrong. 2001. Three simulator tools for teaching computer architecture. Journal on Educational Resources in Computing, 1(4), p. 60–80. DOI:http://dx.doi.org/10.1145/514144.514732

[19] William (Bill) Yurcik. 2002. Special issue on specialized computer architecture simulators that see the present and may hold the future. Journal on Educational Resources in Computing 2, 1 (2002), 1–3. DOI:http://dx.doi.org/10.1145/545197.545198

[20] Linghao Zhang. 2015. Y86-Simulator. Retrieved April 29, 2021 from https://github.com/dnc1994/Y86-Simulator