University of Dublin

TRINITY COLLEGE

Web-based ARM CPU Animation

Rónán Dowling-Cullen
B.A.(Mod.) Computer Science
Final Year Project April 2019
Supervisor: Dr. Jeremy Jones

School of Computer Science and Statistics

O'Reilly Institute, Trinity College, Dublin 2, Ireland

DECLARATION

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university

Name	Date

Abstract

The Web-based ARM CPU Animation is an educational tool designed to help students learn about pipelining concepts and techniques as well as the hardware of the ARM CPU. It demonstrates the pipeline of the ARM9 TDMI CPU and executes ARM assembly language instructions to teach students about pipelining in the context of one of the most common CPU architectures. Taking into account the existing research and best practices surrounding pedagogical animations for computer science education, this animation was designed to be maximally beneficial to students and useful to instructors. Built using VivioJS, the animation runs as HTML5 and JavaScript code allowing it to be published online and used freely by students and instructors around the world.

Acknowledgements

First, I would like to thank Dr. Jeremy Jones for his help and guidance throughout this project. Our discussions about the details of the animation were always insightful and a great learning experience.

I also owe an enormous thank you to my friends and family for their support through the hours, days, weeks and months of work that has culminated in this animation and report.

Contents

List of Figures

List of Tables

1 Introduction

The Web-based ARM CPU Animation is an educational tool designed to help students learn about pipelining and the hardware of the ARM CPU. It demonstrates the pipeline of the ARM9 TDMI CPU and executes ARM assembly language instructions to teach students about pipelining in the context of one of the commonly used CPU architectures.

The initial motivation for this project was the module CS3021: Computer Architecture II, taught to Junior Sophister undergraduate Computer Science students at Trinity College Dublin. The learning objectives of this course state that students should be able to "explain the key concepts behind instruction level pipelining and know how to apply a number of techniques to overcome data, load and control hazards" [SCSS at Trinity College Dublin, 2017b]. Currently, the CS3021 course uses an animation of the MIPS CPU as a learning aid when teaching students about pipelining. Computer Science students at Trinity, however, primarily learn ARM as their assembly language as stated in the brochure published by the School of Computer Science and Statistics [SCSS at Trinity College Dublin, 2017a], .

The project aims to bridge this gap by building an animation of an ARM pipeline that:

- Demonstrates each of the pipelining concepts and techniques in the CS3021 (or any) computer architecture course
- Executes ARM Assembly Language instructions
- Demonstrates ARM specific hardware

The design of this animation was directed by existing research and guidance surrounding pedagogical animations for computer science as well as a review of the existing state of the art in educational animations for teaching pipelining and in animation development tools.

The animation was implemented using VivioJS, a C-like object oriented programming language for creating portable animations that can be hosted online.

To evaluate the animation, it was assessed against the known best practices found in existing research for creating educational animations for computer science and against the existing animation of the MIPS CPU.

The following chapters of this report will outline the background, design, implementation and evaluation of the Web-based ARM CPU Animation, highlighting how the animation was built to achieve each of the goals listed above.

2 Background

The background of the Web-based ARM CPU animation is divided into two parts. The first section of this chapter describes the learning objectives of students using the Web-based ARM CPU Animation. The second section addresses the development tools for creating animations and the existing research and guidance around pedagogical animations for computer science, in particular for animations demonstrating pipelines.

2.1 Learning Objectives of the Animation

2.1.1 CS3021: Computer Architecture II Concepts

As previously stated, the initial motivation for this animation was to help students in their study of the CS3021: Computer Architecture II course taught to undergraduates at Trinity College Dublin. In particular, this project aims to help students deeply understand the concepts of:

- Pipelining
- Data Hazards
- Load Hazards
- Control Hazards

as well as the techniques:

- Pipeline forwarding
- Two Phase Clocking
- Instruction Scheduling
- Delayed Branches
- Branch Prediction

2.1.2 ARM Specific Hardware

An animation designed to demonstrate ARM specific hardware must execute ARM assembly language instructions. Therefore, the ARM instruction set which is executed by this hardware must be understood. The most notable features of this instruction set are explained in [Furber, 2000c, Chapter 2]:

- The load-store architecture
- 3-address data processing instructions (that is, the two source operand registers and the result register are all independently specified)
- conditional execution of every instruction
- the inclusion of very powerful load and store multiple register instructions
- the ability to perform a general shift operation and a general ALU operation in a single instruction that executes in a single clock cycle
- open instruction set extension through the coprocessor instruction set, including adding new registers and data types to the programmer's model
- a very dense 16-bit compressed representation of the instruction set in the Thumb architecture.

The key concept in this list is that the ARM instruction set has a basic format:

```
<Instruction>   <Destination>,<Source1>,<Source2>
```

For example, to add R1 and R2 and store the result in R3 the ARM assembly instruction would be:

```
ADD             R3, R1, R2
```

Additionally, ARM instructions can be extended with a shift operation. For example, to add R1 and R2 (but first shift the value of R2 left by three places) and store the result in R3 the ARM assembly instruction would be:

```
ADD             R3, R1, R2, LSL #3
```

The ARM instruction set is also extended to allow conditional execution of any instruction. To do this, each instruction must have space for a condition that can be checked against the status register flags as well as an S flag to tell the instruction to set the status register flags. For example, to have the previous example set the condition flags the instruction would be:

```
ADDS           R3, R1, R2, LSL #3
```

To have it only be executed if the current status register flags correspond to the equals condition the instruction would be:

```
ADDEQ           R3, R1, R2, LSL #3
```

Lastly, to tell this instruction to set the status register flags and only be executed if the status register flags correspond to the equals condition:

```
        ADDSEQ          R3, R1, R2, LSL #3
```

The standard ARM instruction format is therefore (in general):

```
<Instruction>{S}{<Condition>} <Destination>,<Source1>,<Source2> {<Shift> <Source3>}
```

where {} denotes optional sections.

From this instruction format it is clear that there are two pieces of ARM specific hardware needed for this instruction set:

- The Barrel Shifter as described in [Furber, 2000c, Chapter 4] which allows the inline shifting in ARM instructions
- The Current Program Status Register (CPSR) as described in [Furber, 2000c, Chapter 2] which enables the conditional execution of ARM instructions

Whilst the initial motivation for this animation was the CS3021 course, the pipelining concepts and ARM specific hardware demonstrated are universal. This animation could therefore be used as a teaching aid for any course covering pipelining or ARM hardware.

2.2 Educational Animations for Computer Science

There are a number of elements involved in developing a useful animation to convey the concepts and techniques explained above in the context of the ARM CPU. One element comprises the existing studies and articles discussing best practices in visualisation in computer science education. Another core element is the state of the art in animations attempting to achieve similar goals as well as in development tools suitable for creating educational animations for this purpose.

2.2.1 Visualisation in Computer Science Education

As noted in [Fouh et al., 2012] "Dynamic process is extremely difficult to convey using static presentation media". This has been the motivation for many educational animations, particularly in computer science. Hence, visualisation is a common tool used in computer science courses. Additionally, computer science educators have "a widespread belief that visualisation technology positively impacts learning" as found in surveys of computer science educators undertaken in [Naps et al., 2002]. This belief, however, does not mean all animations are useful or that these educators will use any animation as a teaching aid. The educational impact of a visualisation, therefore, is dependent on:

- the benefit to the learner of the visualisations
- the ease of deployment of the visualisation in the classroom

[Naps et al., 2003]

Factors Affecting the Educational Benefit of Visualisation

While the majority of computer science educators believe that the use of any visualisation improves learning outcomes, this is not substantiated by experimental studies [Hundhausen et al., 2002](as cited by [Naps et al., 2002]). The educational benefit of a given visualisation is dependent on the active engagement of the learner [Naps et al., 2002]. A set of "Best Practices" to improve learner engagement in pedagogical animations are:

1. Provide resources that help learners interpret the graphical representation
2. Adapt to the knowledge level of the user
3. Provide multiple views
4. Include performance information
5. Include execution history
6. Support flexible execution control
7. Support learner-built visualizations
8. Support custom input data sets
9. Support dynamic questions
10. Support dynamic feedback
11. Complement visualizations with explanations

[Naps et al., 2002]

The design of any educational animation for computer science should adhere to the idea of learner engagement as well as the above best practices.

Factors Affecting the Deployment of a Visualisation

The limited time of instructors is one of the greatest barriers that must be overcome by anyone designing an animation for use in teaching and learning computer science [Naps et al., 2003]. The deployment of any animation includes time to find the animation, download it, install it, learn how to use it, integrate it into the course and, finally, teach students how to use it. Therefore, the goal of the designer must be to decrease the time taken to complete any of these stages in order to increase the utility of their animation.

The knowledge of these two components (benefit to learners and ease of deployment for instructors), which affect the educational impact of an animation, was used to direct the design of the Web-base ARM CPU Animation improving its utility.

2.2.2 State of the Art in Pipeling Animations and Animation Development tools

The state of the art for animations of processor pipelines is limited. There is Gem5 [Gem5, 2016], the text based visualisation of a seven stage pipeline, which displays stalls very clearly for a text based visualisation. In five stage pipelines there is Ripes "a graphical 5-stage RISC-V pipeline simulator & assembly editor" [Petersen, 2019], which displays stalls as well as the concept of pipeline forwarding. Unfortunately, these do not meet the learning objectives in the initial motivation for this project as they do not demonstrate all of the pipelining concepts and techniques, execute ARM assembly language or demonstrate ARM specific hardware.

However, the MIPS animation hosted at the VivioJS webpage [Jones, 2019] (see Figure 2.1 below) demonstrates all of the concepts and techniques of CS3021. Developed by Edsko de Vries and Dr. Jeremy Jones in 2003, it is an elegant animation of a simple five stage pipeline. Unfortunately, it requires students to learn an outdated assembly language that they would not otherwise use. As stated in the BA (Mod) in Computer Science/Master in Computer Science brochure [SCSS at Trinity College Dublin, 2017a] computer science students in Trinity College learn ARM Assembly Language. This complicates the deployment of the animation as instructors must add MIPS Assembly Language to their course material in order to use the animation. It also adds a barrier to students learning about pipelining concepts and techniques as they must first learn a new programming language before being able to use the animation.

Animation Development Tools
The choice of development tools for this project was narrowed by the desire to build a portable animation which can be run in browser. This leaves two options, either the animation could be written in HTML and JavaScript or in a language which compiles into these languages. As is explained further in the design chapter of this report, this animation was written in VivioJS. VivioJS is an object oriented, C-like programming language which compiles into "HTML5 and JavaScript making [its animations] portable across browsers, operating systems and machine architectures" [Jones, 2019]. It is an extension of Vivio "a system that makes it easier to create interactive reversible 2D vector based e-learning animations for the WWW" [Jones, 2004], which was created by Dr. Jeremy Jones. See an example of VivioJS below (Figure

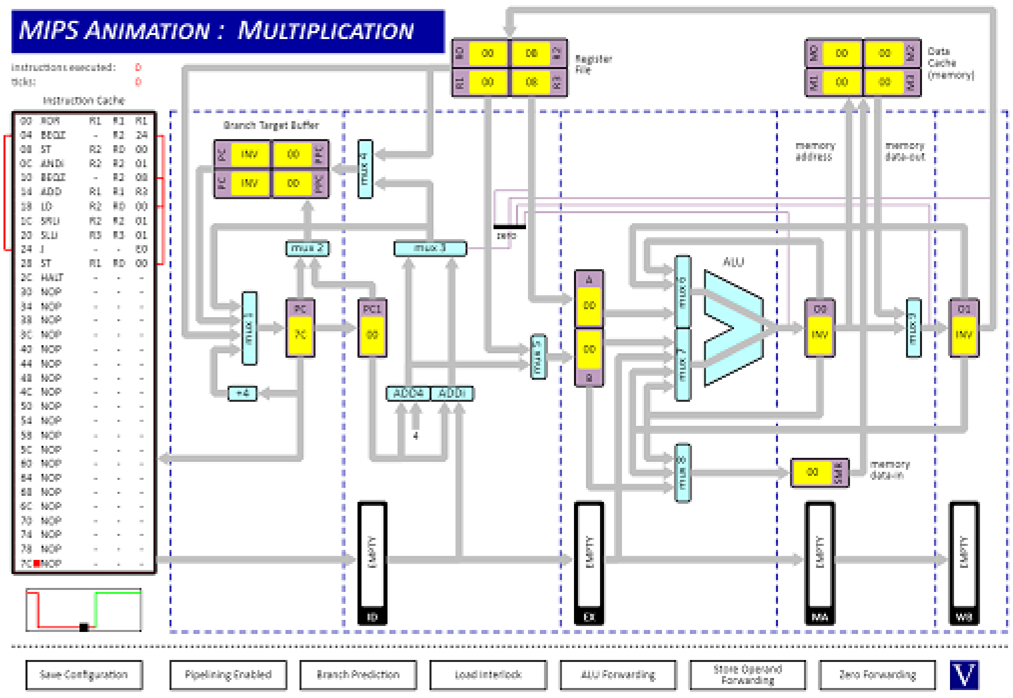
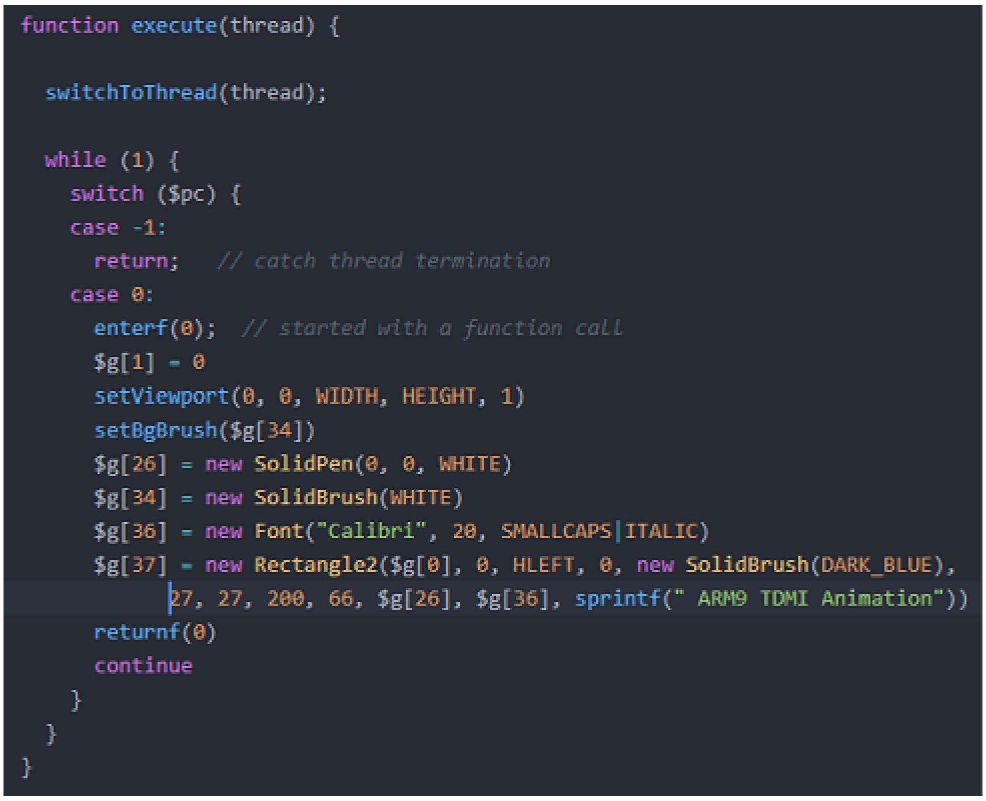


Figure 2.1: Existing MIPS animation.

2.2), followed by the generated JavaScript (Figure 2.3) calling upon functions in the Vivio.js library (which is hosted in the same web page) and then the actual animation as displayed by a web browser (Figure 2.4).

```
Font fTitle = Font("Calibri", 20, SMALLCAPS | ITALIC);
Rectangle title = Rectangle2(0, HLEFT, 0, SolidBrush(DARK_BLUE), 27, 27, 200.0,
                             66, whitePen, fTitle, sprintf(" ARM9 TDMI Animation"));
```

Figure 2.2: Sample VivioJS Code



```
function execute(thread) {

  switchToThread(thread);

  while (1) {
    switch ($pc) {
    case -1:
      return;    // catch thread termination
    case 0:
      enterf(0);  // started with a function call
      $g[1] = 0
      setViewport(0, 0, WIDTH, HEIGHT, 1)
      setBgBrush($g[34])
      $g[26] = new SolidPen(0, 0, WHITE)
      $g[34] = new SolidBrush(WHITE)
      $g[36] = new Font("Calibri", 20, SMALLCAPS|ITALIC)
      $g[37] = new Rectangle2($g[0], 0, HLEFT, 0, new SolidBrush(DARK_BLUE),
            27, 27, 200, 66, $g[26], $g[36], sprintf(" ARM9 TDMI Animation"))
      returnf(0)
      continue
    }
  }
}
```

Figure 2.3: Sample Generated JavaScript Code

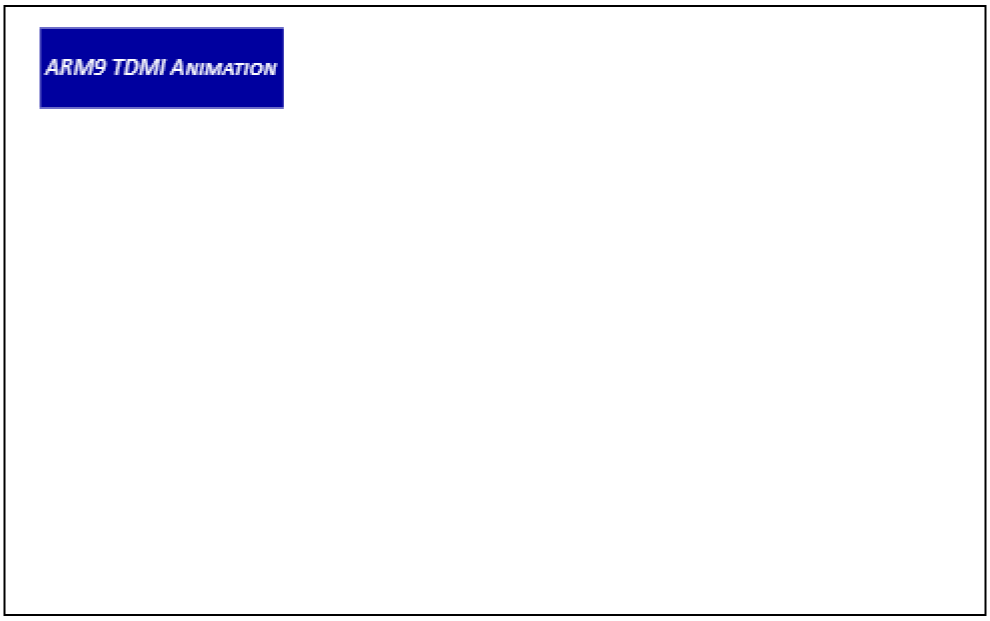


Figure 2.4: Sample Animation Displayed in Browser

2.3 Summary

The Web-based ARM CPU Animation seeks to help teach students about the core concepts and techniques surrounding pipelining as well as ARM specific hardware. The standard ARM instruction format is (in general):

```
<Instruction>{S}{<Condition>} <Destination>,<Source1>,<Source2> {<Shift> <Source3>}
```

The common limitations of educational animations are that they either do not benefit learners or are too difficult to deploy for instructors. The Web-based ARM CPU Animation was written in VivioJS, an object oriented language that compiles into HTML5 and JavaScript making it extremely portable. The existing MIPS animation meets all the desired learning objectives of CS3021 but, unfortunately, it requires students to learn an extra assembly language which impedes their learning about pipelines. The ideal animation would cover all the pipelining learning objectives, have learning objectives that can help students around the world, be easy to deploy for instructors and animate an ARM architecture allowing students to write programs in a language they already know. The next chapters show the design and implementation of such an animation.

3 Design

This section will outline the design decisions made before and during the implementation of the Web-based ARM CPU Animation and explain the motivation behind each of these. The decisions are covered in five subsections: Instruction Set, Architecture, Choice of Programming Language, Existing MIPS Animation and Learning Aids.

3.1 Instruction Set

The first decision about the instruction set concerned how the instructions would be entered. The style of the existing MIPS CPU animation was not changed, which will allow students currently using the MIPS animation to easily navigate the new animation built in this project. In the MIPS animation, the instructions are entered by clicking on the ARM instruction mnemonic in each line of the instruction cache to cycle through the implemented instructions until the desired instruction is shown. Each operand necessary for that instruction can then be selected and operands not relevant to that instruction cannot.

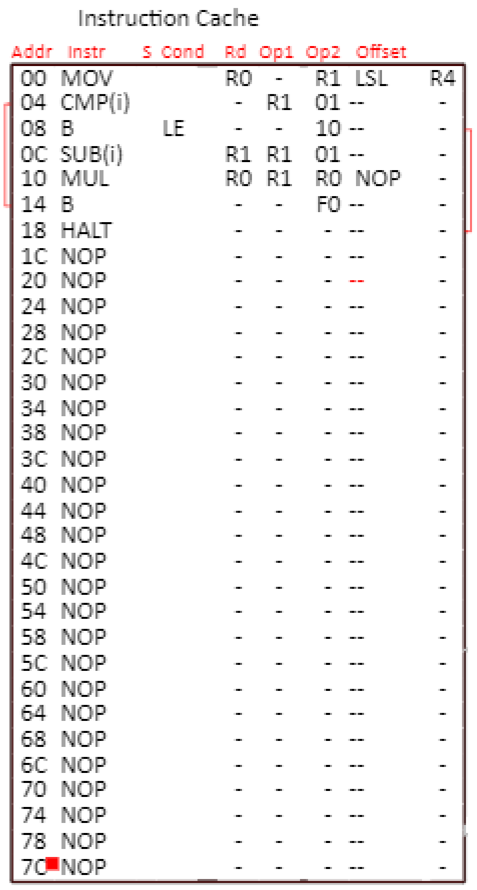
This style was extended to include the optional shift and rotate extensions on each instruction (as explained later in this section), see Figure 3.1 on the next page.

For the same reason the practice of using separate mnemonics for instructions operating on registers only, and on registers and immediate values, was maintained from the MIPS animation. For example, adding two registers in the animation is entered as:

```
ADD     R1,R2,R3
```

whereas adding a register and an immediate value is entered as:

```
ADD(i)  R1,R2,#3
```



Instruction Cache

Addr	Instr	S	Cond	Rd	Op1	Op2	Offset	
00	MOV			R0	-	R1	LSL	R4
04	CMP(i)			-	R1	01	--	-
08	B		LE	-	-	10	--	-
0C	SUB(i)			R1	R1	01	--	-
10	MUL			R0	R1	R0	NOP	-
14	B			-	-	F0	--	-
18	HALT			-	-	-	--	-
1C	NOP			-	-	-	--	-
20	NOP			-	-	-	--	-
24	NOP			-	-	-	--	-
28	NOP			-	-	-	--	-
2C	NOP			-	-	-	--	-
30	NOP			-	-	-	--	-
34	NOP			-	-	-	--	-
38	NOP			-	-	-	--	-
3C	NOP			-	-	-	--	-
40	NOP			-	-	-	--	-
44	NOP			-	-	-	--	-
48	NOP			-	-	-	--	-
4C	NOP			-	-	-	--	-
50	NOP			-	-	-	--	-
54	NOP			-	-	-	--	-
58	NOP			-	-	-	--	-
5C	NOP			-	-	-	--	-
60	NOP			-	-	-	--	-
64	NOP			-	-	-	--	-
68	NOP			-	-	-	--	-
6C	NOP			-	-	-	--	-
70	NOP			-	-	-	--	-
74	NOP			-	-	-	--	-
78	NOP			-	-	-	--	-
7C	NOP			-	-	-	--	-

Figure 3.1: Instruction Cache

The next decision concerned which instructions to support. The definitions of all instructions implemented were taken from [Furber, 2000c, Chapter 5]. The full list of instructions implemented can be found in Table A1.1 of the Appendix. For clarity, the following subheadings separate the instructions into their functional categories.

3.1.1 Branches

There are two types of branch instruction in ARM assembly language - branch, and branch and link. The branch instruction (B) simply branches to a label. The branch and link (BL) instruction branches to a label but also stores the current value of the program counter in the link register. This allows execution to return to the point branched from after executing some instructions and is primarily used for branching to subroutines. Both branch instructions were included as understanding the stalls due to branches is a fundamental part of a students understanding of pipeline execution on any pipelined processor. As well as this, branch and link is a key instruction in the ARM assembly language as it helps with the design of subroutines.

The exchange versions (BX and BLX) of the above branch instructions were not included as they are used to switch between the ARM and Thumb instructions sets and the Thumb instruction set is not implemented in the animation as explained under the Functional Categories Not Included subheading below.

3.1.2 Data Processing

There are sixteen data processing instructions which perform logical, arithmetic, comparative (to update the current program status register) and move operations on registers or immediate values, see Table 3.1 below for a list of these instructions and their definitions.

Table 3.1: ARM Data Processing instructions [Furber, 2000c, Chapter 5]

Opcode	Mnemonic	Meaning	Effect
0000	AND	Logical bit-wise AND	Rd := Rn AND Op2
0001	EOR	Logical bit-wise exclusive OR	Rd := Rn EOR Op2
0010	SUB	Subtract	Rd := Rn - Op2
0011	RSB	Reverse subtract	Rd := Op2 - Rn
0100	ADD	Add	Rd := Rn + Op2
0101	ADC	Add with carry	Rd := Rn + Op2 + C
0110	SBC	Subtract with carry	Rd := Rn - Op2 + C - 1
0111	RSC	Reverse subtract with carry	Rd := Op2 - Rn + C - 1
1000	TST	Test	Sccon Rn AND Op2
1001	TEQ	Test equivalence	Sec on Rn EOR Op2
1010	CMP	Compare	Sec on Rn - Op2
1011	CMN	Compare negated	Sec on Rn + Op2
1100	ORR	Logical bit-wise OR	Rd := Rn OR Op2
1101	MOV	Move	Rd := Op2
1110	BIC	Bit clear	Rd := Rn AND NOT Op2
1111	MVN	Move negated	Rd := NOT Op2

All of the ARM data processing instructions were implemented in the animation as they are all commonly used in normal ARM assembly programming.

3.1.3 Multiplication

The standard multiplication instruction (MUL) was included as it is useful for students to understand stalls around multiplication in the five stage pipeline of the ARM9. This is explained in greater detail in the implementation section of this report. The non-standard multiplication instructions (MLA, UMULL, UMLAL, SMULL and SMLAL) were not included to simplify the instruction set as they are not commonly used instructions in ARM Assembly programming and their function can be completed with a short sequence of the instructions included in the animation.

3.1.4 Conditional Execution

As stated in [Furber, 2000c, Chapter 5] "An unusual feature of the ARM instruction set is that every instruction (with the exception of certain v5T instructions) is conditionally executed". This means that any instruction can be prevented from executing based on the flags in the current program status register (CPSR). As this is an unusual but powerful feature of the ARM instruction set and conditional branching is key to the utility of any assembly language, it was included in the animation.

There are eighteen possible conditions that can be checked on any instruction in ARM assembly language (nineteen if the default which is equivalent to always (AL) is included). These can be seen in Table A1.2 in the appendix. All nineteen are included in the instruction set of the animation as they each perform a unique function and they are an important part of a students understanding of pipelining in ARM.

3.1.5 Data Transfer

The two standard ARM instructions for loading from memory (LDR) and storing to memory (STR) were included because ARM is a load and store architecture making these instructions central to its execution.

The address being loaded from or stored to in both of these instructions can be offset in four ways:

- Adding a register to the address

  ```
  LDR R1,[R0,R3]
  ```

  (Where R1 is the destination register, R0 contains the base memory address and R3 contains the offset.)

- Adding an immediate value to the address

  ```
  LDR R1,[R0,#3]
  ```

  (Where R1 is the destination register, R0 contains the base memory address and 3 is the offset.)

- Adding a register that has been shifted or rotated by a register to the address

  ```
  LDR R1,[R0,R3,LSL R4]
  ```

(Where R1 is the destination register, R0 contains the base memory address, R3 is the offset register, LSL is the shift instruction and R4 contains the amount R3 is shifted by.)

- Adding a register that has been shifted or rotated by an immediate value to the address

```
LDR R1,[R0,R3,LSL #4]
```

(Where R1 is the destination register, R0 contains the base memory address, R3 is the offset register, LSL is the shift instruction and 4 is the amount R3 is shifted by.)

All four of these offsetting extensions were included as they are often used to reduce the number of instructions needed for an indexed memory access (without inducing extra stalls). This makes it an important feature for students to understand in the context of pipelining.

Multiple register data transfer instructions such as load multiple (LDM) and store multiple (STM) were not included due to the variable number of registers that can be used as operands in the instructions. This is not possible with the current design of the instruction cache into which instructions are to be entered. The functionality of these instructions, however, can be implemented by combining multiple load or store instructions.

3.1.6 Shift and Rotate

The four shift and rotate instructions (see Table 3.2 below) which can be applied to operand two of almost any instruction were included in the animation because "The ability to perform a general shift operation and a general ALU operation in a single instruction that executed in a single clock cycle ...[is one of]... the most notable features of the ARM instruction set" [Furber, 2000c, Chapter 2]. This makes them a useful tool for students seeking to reduce the number of stalls in the pipeline.

Table 3.2: ARM shift and rotate instructions

Mnemonic	Meaning	Effect
LSL	Logical Shift Left	Shift the register left introducing zeros on the right
LSR	Logical Shift Right	Shift the register right introducing zeros on the left
ASR	Arithmetic Shift Right	Shift the register right and maintain the sign by introducing ones or zeros on the left
ROR	Rotate Right	Rotate the register right

In the ARM CPU, stand alone shift and rotate instructions (eg. LSL R1,R2,R3 shifting R2 by R3 and putting the result in R1) are synthesized into move instructions with the source register being shifted or rotated (the example above would become MOV R1,R2,LSL R3).

This feature was added to show students the conversion of pseudo instructions into other instructions and how this affects the pipeline.

3.1.7 Functional Categories Not Included

Software interrupt instructions (SWI) are used for calls to the operating system [Furber, 2000c, Chapter 5]. These were not included as this is just an animation so there is no operating system.

Coprocessor instructions (normally used to operate the coprocessor which controls the on-chip functions like the cache and the memory management unit [Furber, 2000c, Chapter 5]) were not included as the animation does not have any coprocessors.

The 16-bit THUMB instruction set was not included for simplicity as THUMB Assembly Language is much less common than ARM Assembly Language.

Status register transfer instructions (primarily used to maintain the context of one thread while another thread executes) were not included because multi-threaded programming is beyond the scope of this project.

3.2 Architecture

In order to create a useful animation that helps students learn about pipelining concepts and techniques, the architecture and the hardware chosen to be animated was carefully considered. This section will explain each of those considerations and the decisions made as a result.

3.2.1 ARM9 TDMI

The first choice that needed be made was which CPU pipeline architecture should be animated. The motivation for this project was to create an animation to help students learn about pipelining concepts and techniques, that will run programs in the programming language they already know, ARM Assembly Language. Therefore, the pipeline animated must be that of an ARM CPU.

Next, the specific version of the ARM CPU needed to be chosen. The ARM9 TDMI was selected (TDMI stands for Thumb, Debugger, Multiplication and In-circuit emulator, each of its extensions). It was chosen as its pipeline supports many of the desired techniques (including pipeline forwarding which is not supported on the earlier ARM7 CPUs and two phase clocking, see Figure 3.2 below) while not being overly complicated (having only a five stage pipeline compared to the six stages of the ARM10, see Figure 3.3 below). The ARM9 TDMI architecture covers most of the techniques in the learning objectives of this animation but it does not support them all. To demonstrate the remaining techniques in the learning objectives of CS3021 this animation includes hardware beyond the ARM9 TDMI architecture. The following sections outline where these extensions occur.

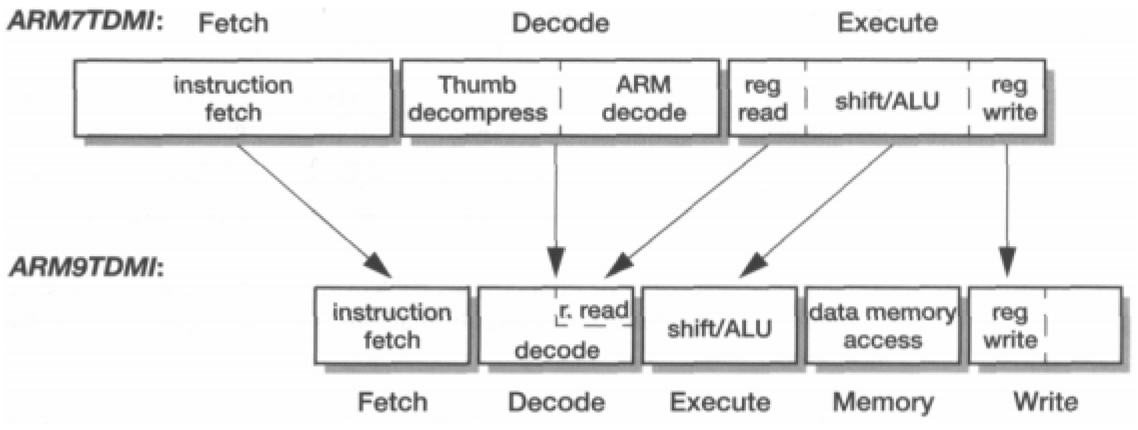


Figure 3.2: ARM7TDMI and ARM9TDMI pipeline comparison
[Furber, 2000b]

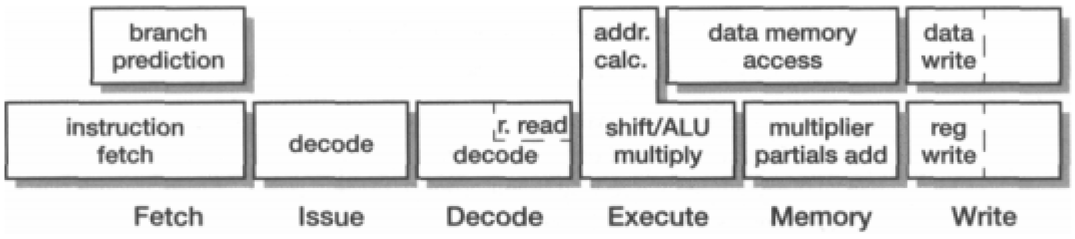


Figure 3.3: The ARM10TDMI pipeline
[Furber, 2000a]

3.2.2 Branch Prediction

Branch prediction is not supported in the ARM9 TDMI, but it is supported in the ARM10, see figure 3.3 above. However, it is one of the techniques in the learning objectives from the motivation of this project. Branch prediction functionality was included, mimicking the functionality of the ARM10 CPU, with the caveat that this functionality does not exist in the ARM9. Figure 3.4 shows how the branch target buffer is used to predict branches. It is populated with a predicted branch from the instruction at 0x14 to 0x04. This feature can be toggled on and off allowing students to compare the stalls induced and the clock cycles needed to execute the same code with and without the feature.

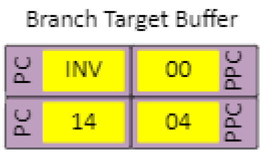


Figure 3.4: Populated Branch Target Buffer Animation

3.2.3 Delayed Branches

"Some RISC architectures (though not the ARM) define that the instruction following the branch is executed whether or not the branch is taken. This technique is known as the delayed branch." [Furber, 2000c, Chapter 1]. Delayed branching is also one of the techniques in the learning objectives from the motivation of this project. Therefore, it was included in the ARM9 animation with the caveat that ARM does not support this. This feature can be toggled on and off allowing students to compare the stalls induced and the clock cycles needed to execute the same code with and without the feature.

3.2.4 Current Program Status Register

The current program status register (CPSR) in ARM is checked when performing conditional operations. It was important to visualise this because (as explained above) conditional branching is key in any assembly language and conditional execution of non-branching instructions is one of the unusual yet powerful features of ARM (as it can be used to avoid pipeline stalls by removing the need for branch instructions). The CPSR in the ARM9 CPU is a full 32-bit register, however, for simplicity only the flags needed for conditional execution (**N**egative (N), **Z**ero (Z), **C**arry (C) and O**V**erflow (V)) were animated, see figure 3.5 below.

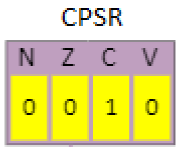


Figure 3.5: Current Program Status Register (CPSR) Animation

3.2.5 Stack Animation

The use of stacks is a fundamental part of ARM Assembly programming. For this reason an animation of a full descending stack implementation was added. This helps students understand stack based programming. A full descending implementation was chosen as this is the most common implementation used when programming in ARM Assembly Language, see figure 3.6 below.

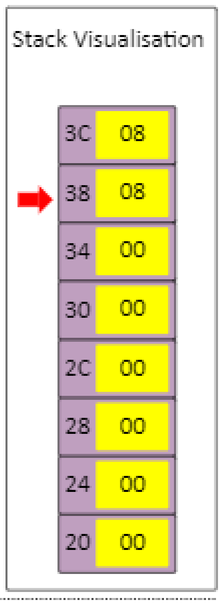


Figure 3.6: Full Descending Stack Animation

3.2.6 Hardware Extensions

The existing MIPS animation has four registers and four memory locations, see figures 3.7 and 3.8 below.

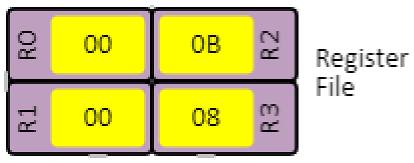


Figure 3.7: MIPS Animation: Register File [Jones, 2019]

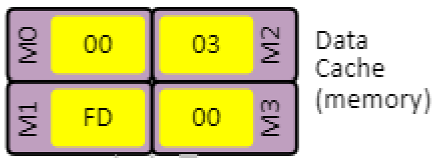
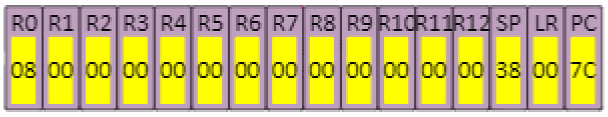


Figure 3.8: MIPS Animation: Memory [Jones, 2019]

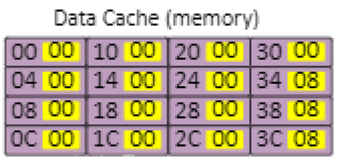
The ARM9 has sixteen registers so the register file in the ARM9 animation was extended to include all sixteen, see figure 3.9 below. This was particularly important as the highest three ARM9 registers R13, R14 and R15 (the stack pointer (SP), the link register (LR) and the program counter (PC) respectively) are crucial to a students understanding of ARM stack implementations and flow control. This extension along with the CPSR animation allows the Web-based ARM CPU Animation to support all programs that can be executed in user-mode on the ARM9, because "When writing user-level programs, only the 15 general-purpose 32-bit registers (r0 to r14), the program counter (r15) and the current program status register (CPSR) need be considered." [Furber, 2000c, Chapter 2]



R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	SP	LR	PC
08	00	00	00	00	00	00	00	00	00	00	00	00	38	00	7C

Figure 3.9: ARM9 TDMI Animation: Register File

It was decided to also extend the memory in the ARM9 Animation from four locations to sixteen to give space for the stack implementation, see figure 3.10.



Data Cache (memory)

00	00	10	00	20	00	30	00
04	00	14	00	24	00	34	08
08	00	18	00	28	00	38	08
0C	00	1C	00	2C	00	3C	08

Figure 3.10: ARM9 TDMI Animation: Memory

3.3 Choice of Programming Language

The ARM9 animation was designed to be as portable as possible. It can be used on any device with a browser and hosted on any website as it is run as HTML5 and JavaScript. The animation was not written in HTML5 and JavaScript, however, it was written in VivioJS. There are four main reasons for this:

- VivioJS compiles into HTML5 and JavaScript giving it the same portability as if it was written in them.
- VivioJS supports flexible execution control, this allows learners to run the animation forwards and backwards as well as allowing them to step through the program.
- JavaScript is a single threaded programming language, making it difficult to animate the five different stages of the pipeline. VivioJS provides pseudo parallel constructs which remove most of this complication.
- The existing MIPS animation is written in VivioJS so writing the ARM9 in the same language allows for similar execution control of the animation making it easier for students using the MIPS animation to use the ARM9 animation.

3.4 Existing MIPS Animation

Having been a student in the CS3021 course I can say that the existing MIPS animation (see figure 2.1) is an excellent teaching aid. It is designed to demonstrate the pipelining concepts and techniques taught in CS3021: Computer Architecture II and it has a five stage pipeline similar to that of the ARM9. For these reasons it was decided to convert the existing MIPS animation to ARM and then build upon it as opposed to starting from scratch. This allowed more complex features like conditional execution using the CPSR to be achieved as well as allowing students currently using the MIPS animation to easily switch to the ARM9 animation.

3.5 Learning Aids

A key part of understanding pipelining comes from comparing the number of clock cycles needed to execute a given program. Accordingly, the number of instructions executed and clock ticks elapsed is displayed in the top left corner similar to the MIPS animation (see figure 3.11 below).

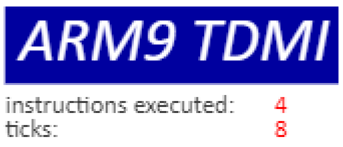


Figure 3.11: ARM9 TDMI Animation: Instruction and Clock Tick Counts

The concepts of pipelining can be quite daunting to new computer science students. To minimise this a few features have been added to help students when they first start using the animation.

Firstly, a help screen similar to that of the MIPS animation is shown when starting the animation to explain how to run the animation, load a sample program, enter instructions, and change register values, see figure 3.12 below.

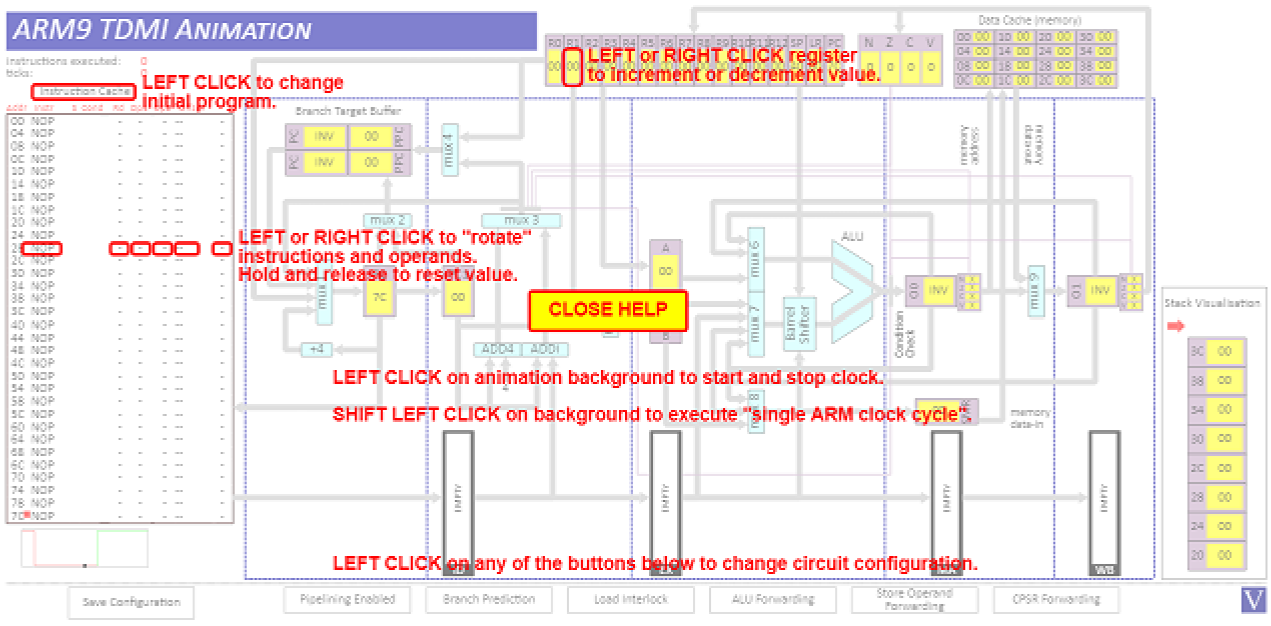

ARM9 TDMI ANIMATION
LEFT CLICK to change initial program.
LEFT or RIGHT CLICK register to increment or decrement value.
LEFT or RIGHT CLICK to "rotate" instructions and operands. Hold and release to reset value.
CLOSE HELP
LEFT CLICK on animation background to start and stop clock.
SHIFT LEFT CLICK on background to execute "single ARM clock cycle".
LEFT CLICK on any of the buttons below to change circuit configuration.
Save Configuration
Pipelining Enabled
Branch Prediction
Load Interlock
ALU Forwarding
Store Operand Forwarding
CPSR Forwarding
Stack Visualisation

Figure 3.12: ARM9 TDMI Animation: Help Screen

Secondly, sample programs which demonstrate each of the concepts and techniques in the learning objectives of the CS3021: Computer Architecture II course are loaded in the animation. These can be run without the students having to enter any instructions themselves.

3.6 Summary

This animation has been designed to achieve the goals drawn from the initial motivation of this project:

- To demonstrate each of the pipelining concepts and techniques in the learning objectives of the CS3021: Computer Architecture II Course.
- To allow students to enter ARM Assembly Language programs to see these techniques in action.
- To demonstrate ARM hardware to give students a deeper understanding of the CPU most relevant to them.

This was achieved by:

- Carefully selecting a subset of the ARM instruction set that can be executed in the animation such that all the concepts and techniques can be demonstrated whilst keeping the instruction cache simple for new students.
- Choosing an architecture to animate that allows the students to enter ARM Assembly programs and supports hardware which demonstrates most of the techniques desired

- Adding extra functionality and hardware that does not exist on the ARM9 TDMI CPU (delayed branches and branch prediction) to ensure that all pipelining concepts and techniques are supported.

The background section of this report explains that, in order to design and implement the most useful pedagogical animation, it must be educationally beneficial to learners and should follow the set of best practices. The design of the Web-based ARM CPU Animation was directed by this knowledge and it will be assessed against these best practices in the Evaluation chapter of this report.

The background section also notes that for an educational animation to be useful it must be quick to deploy. This was achieved by designing the implementation of the animation such that it can be hosted on a website meaning it can be run in browser. This requires no download or installation and can be run on any device with a browser, greatly decreasing the time needed for an instructor to deploy the animation.

4 Implementation

The Web-based ARM CPU animation consists of approximately five thousand lines of VivioJS code, including one hundred and seventeen functions and ten different classes. This compiles into nearly five thousand lines of JavaScript which is displayed on a HTML5 canvas in browser when run. This chapter first describes the pipeline execution and implementation of the pipelining concepts described in the background chapter. Next, the chapter explains the path to completion of the Web-based ARM CPU Animation, including the conversion of the existing MIPS animation, extension of the existing hardware in the animation and addition of ARM specific hardware to execute ARM instructions.

4.1 Pipeline Implementation

This section of the chapter will explain how the pipeline in the Web-based ARM CPU Animation was implemented and how the different concepts and techniques outlined in the learning objectives of the CS3021 course are demonstrated using VivioJS code.

4.1.1 Pipeline Execution

The ARM9 TDMI pipeline consists of five stages:

- Instruction Fetch (IF)
- Instruction Decode (ID)
- Execute (EX)
- Memory Access (MA)
- Write Back (WB)

This subsection describes the implementation of each stage of the pipeline. In discussing each of the stages it is explained how the Web-based ARM CPU Animation demonstrates each of the pipelining concepts and techniques in the CS3021 course. The execution of instructions by this pipeline is implemented in the "execution.vin" file in Appendix 2.

Instruction Fetch

The IF stage of the pipeline is responsible for fetching the next instruction from the Instruction Cache. This is only executed if the pipeline is not stalled for that clock cycle. Once the instruction is fetched the Program Counter (PC) is checked against the saved predicted branches in the branch target buffer, see Figure 3.4. If the PC is in the branch target buffer then the predicted PC is set as the next PC (the rest of the branch target buffer operation is handled in the instruction decode phase).

The IF stage also manages the changes a compiler normally makes to ARM programs before they are executed. This is where shift and rotate instructions are transformed into move instructions with an inline shift or rotate, as explained in the Design chapter of this report, see the code snippet in Figure 4.1 below.

```
//Convert shift ops to move ops
if((regID.vIns >= LSL && regID.vIns <= ROR) || (regID.vIns >= LSLi && regID.vIns <= RORi)){
     regID.vIns2 = regID.vIns;
     regID.vIns = MOV;
     regID.vRs3 = regID.vRs2;
     regID.vRs2 = regID.vRs1;
}
```

Figure 4.1: ARM9 TDMI Animation: Shift and Rotate Conversion VivioJS Code

Instruction Decode

The ID stage is responsible for getting the operands (if they are registers) from the register file to be passed to the Arithmetic Logic Unit (ALU) and used in the EX stage, see Figure 4.2 showing the registers for an add instruction being retrieved from the register file. The ID stage also demonstrates part of the two phase clocking technique by showing that the register file is only read on the second half of the clock cycle.

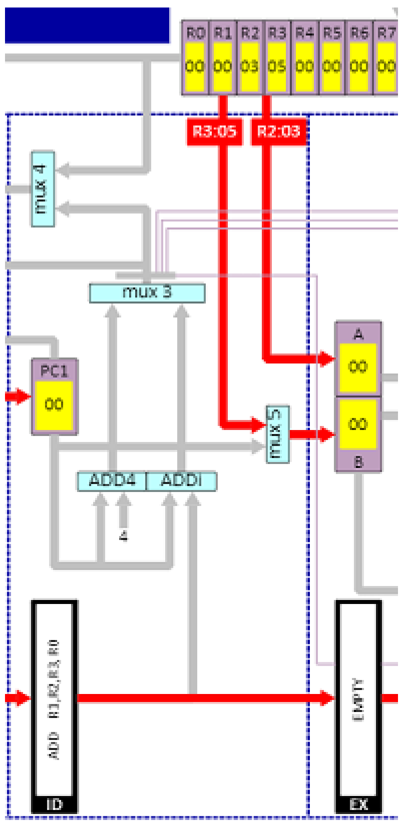


Figure 4.2: ARM9 TDMI Animation: Instruction Decode of Add Instruction

The ID stage is also responsible for the second half of the branch prediction functionality. If the current PC is in the branch target buffer then the PC is set to predicted value on the next clock cycle. If the branch was not in the branch target buffer, or the branch was predicted incorrectly, then a one clock cycle stall is induced in the pipeline to allow the correct instruction to be loaded from the instruction cache. If a branch is correctly predicted or not in the branch target buffer and conditionally not executed then no stall is induced. If the instruction is a branch and it was taken then the previous PC and the address it branched to are added as a pair to the branch target buffer. As most branches are usually taken every time except the last time, or conditionally not taken every time except the last time (think about loops), predicting that a branch will go to the same address as last time is correct most of the time and decreases the average number of stalls in a program, see Figures 4.3 and 4.4 which show a branch instruction passing through the pipeline, updating the branch target buffer. This branch prediction functionality is one of the learning objectives of the animation and can be toggled on and off to demonstrate this technique.

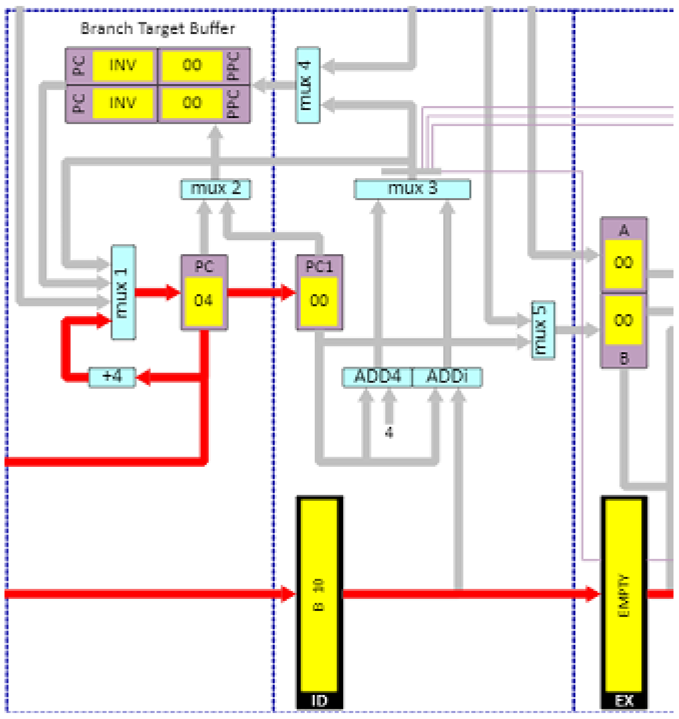


Figure 4.3: MIPS Animation: Branch in Pipeline Before Branch Target Buffer Update

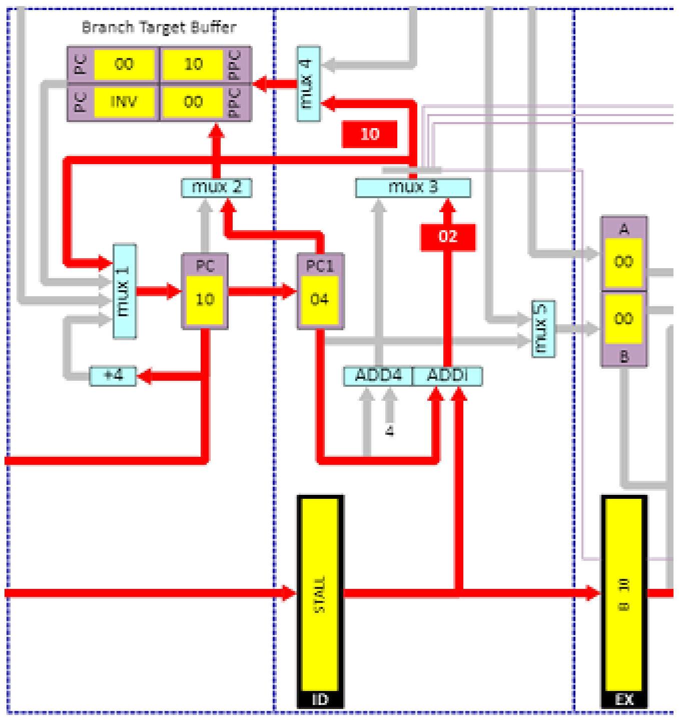


Figure 4.4: ARM9 TDMI Animation: Branch in Pipeline After Branch Target Buffer Update

If branch prediction is toggled off and delayed branching (another of the desired learning objectives of the animation) is toggled on then the instruction immediately after the branch is always executed.

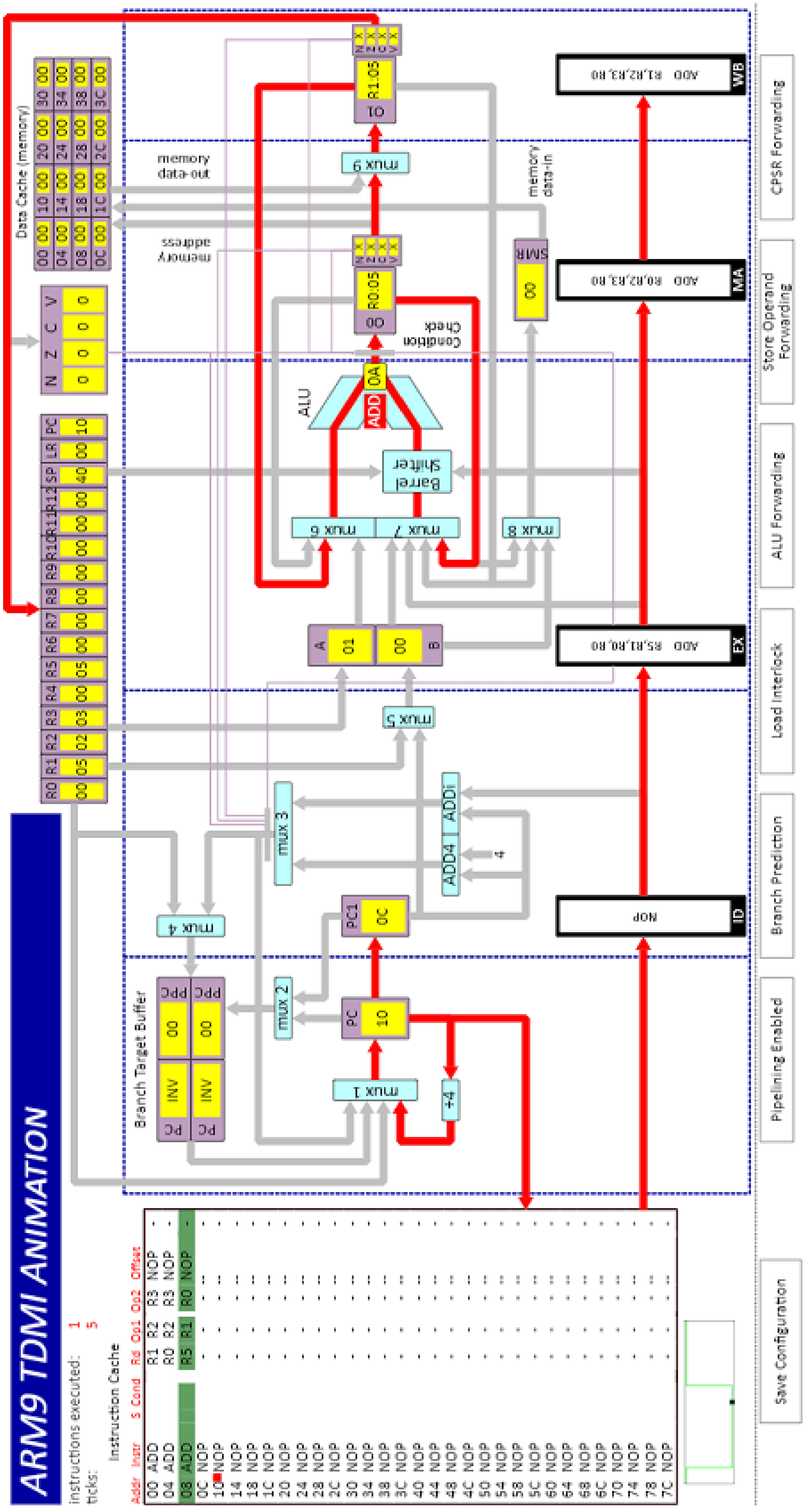
The techniques of branch prediction and delayed branches demonstrate how control stalls can be overcome.

Execute

The EX stage of the pipeline is responsible for executing each instruction. It can perform pipeline forwarding (another one of the learning objectives of the animation) if the hardware to do this is toggled on. This demonstrates to students how data stalls can be overcome. See Figure 4.5 on the following to see how pipeline forwarding is shown in the animation when an add instruction that is data dependent on the two previous instructions is executed.

The EX phase also demonstrates the ARM specific multiplication stalls. The ALU in the ARM9 TDMI implements an 8-bit Booth's algorithm to multiply numbers together. This induces a one clock cycle stall for every eight bits of the operands being multiplied that are not all ones or all zeros and preceded by all ones or all zeros [Furber, 2000c, Chapter 4], see Figures 4.6 and 4.7 on the following pages showing a stall induced in the pipeline due to a multiplication operation.

The EX stage also loads immediate values from the instruction if there are any.



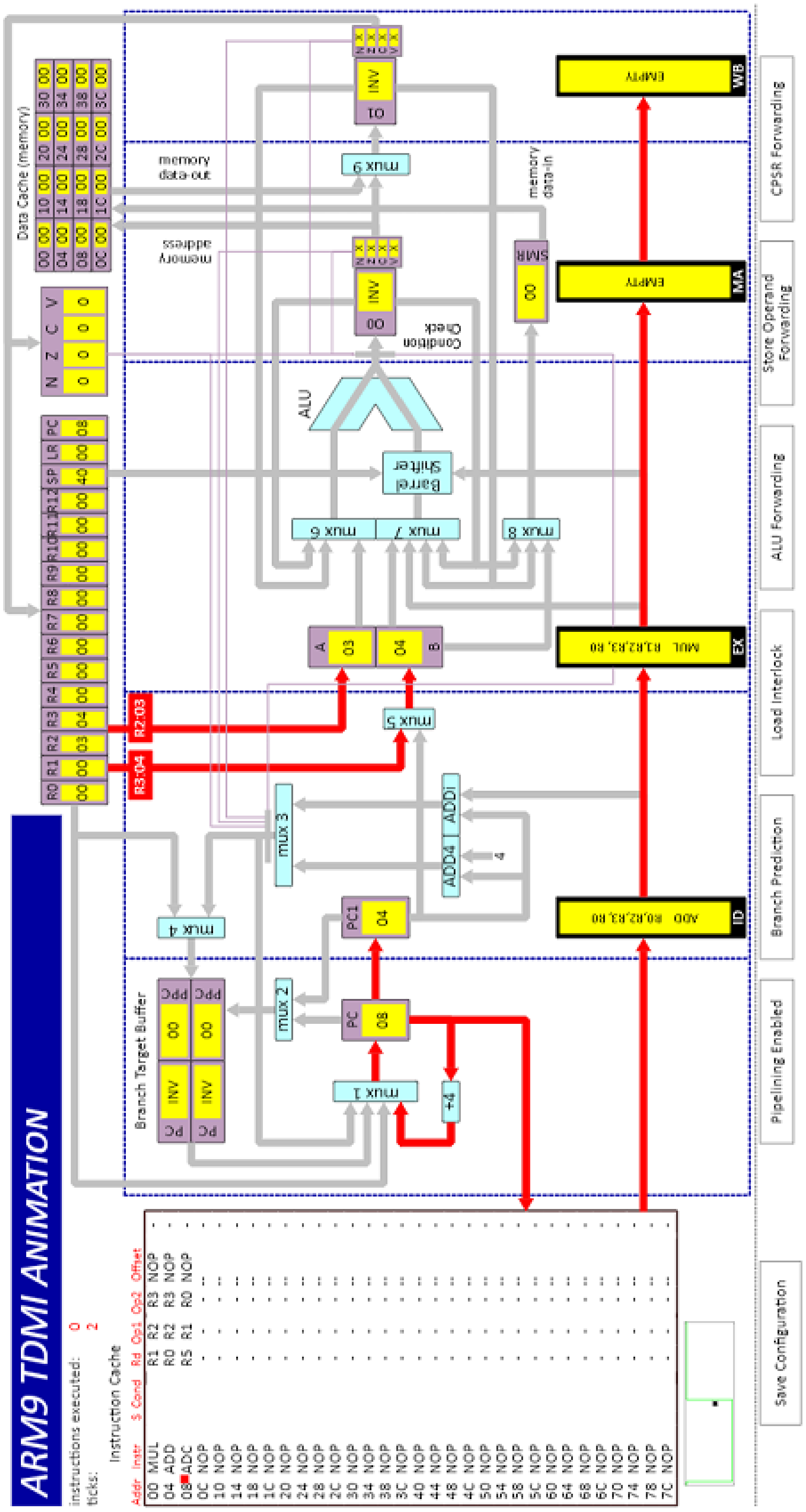
```
Addr Instr  S Cond  Rd Op1 Op2 Offset
00   ADD            R1 R2  R3  NOP  -
04   ADD            R0 R2  R3  NOP  -
08   ADD            R5 R1  R0  NOP  -
0C   NOP            -  -   -   --   -
10   NOP            -  -   -   --   -
14   NOP            -  -   -   --   -
18   NOP            -  -   -   --   -
1C   NOP            -  -   -   --   -
20   NOP            -  -   -   --   -
24   NOP            -  -   -   --   -
28   NOP            -  -   -   --   -
2C   NOP            -  -   -   --   -
30   NOP            -  -   -   --   -
34   NOP            -  -   -   --   -
38   NOP            -  -   -   --   -
3C   NOP            -  -   -   --   -
40   NOP            -  -   -   --   -
44   NOP            -  -   -   --   -
48   NOP            -  -   -   --   -
4C   NOP            -  -   -   --   -
50   NOP            -  -   -   --   -
54   NOP            -  -   -   --   -
58   NOP            -  -   -   --   -
5C   NOP            -  -   -   --   -
60   NOP            -  -   -   --   -
64   NOP            -  -   -   --   -
68   NOP            -  -   -   --   -
6C   NOP            -  -   -   --   -
70   NOP            -  -   -   --   -
74   NOP            -  -   -   --   -
78   NOP            -  -   -   --   -
7C   NOP            -  -   -   --   -
```

R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	SP	LR	PC
00	05	02	03	00	05	00	00	00	00	00	00	00	40	00	10

N	Z	C	V
0	0	0	0

00	00	10	00	20	00	30	00
04	00	14	00	24	00	34	00
08	00	18	00	28	00	38	00
0C	00	1C	00	2C	00	3C	00

Figure 4.5: ARM9 TDMI Animation: Pipeline Forwarding



```
Addr Instr S Cond Rd Op1 Op2 Offset
00 MUL            R1 R2  R3  NOP
04 ADD            R0 R2  R3  NOP
08 ADC            R5 R1  R0  NOP
0C NOP
10 NOP
14 NOP
18 NOP
1C NOP
20 NOP
24 NOP
28 NOP
2C NOP
30 NOP
34 NOP
38 NOP
3C NOP
40 NOP
44 NOP
48 NOP
4C NOP
50 NOP
54 NOP
58 NOP
5C NOP
60 NOP
64 NOP
68 NOP
6C NOP
70 NOP
74 NOP
78 NOP
7C NOP
```

Figure 4.6: ARM9 TDMI Animation: Multiplication Before Stall

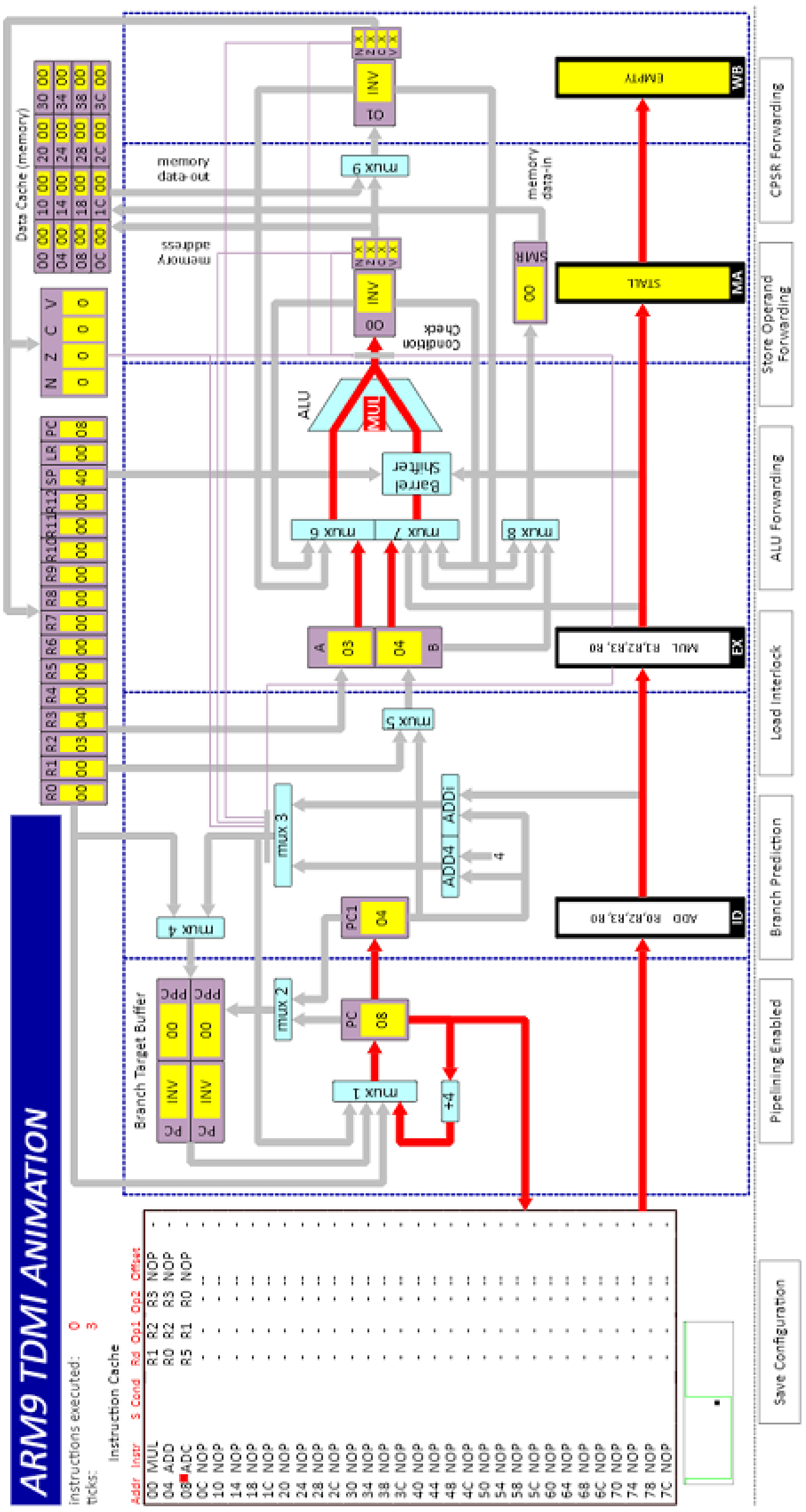


Figure 4.7: ARM9 TDMI Animation: Multiplication After Stall

Memory Access

The MA phase of the pipeline is responsible for load and store instructions accessing memory. It shows how instruction scheduling (another one of the techniques in the learning objectives of the animation) is necessary as the pipeline induces a stall when the instruction immediately after a load instruction requires the value loaded from memory. As this will not be in the first forwarding register, it must wait for the load instruction to finish the MA stage before the value can be forwarded, see Figure 4.8 below in which the animation has induced a stall in the pipeline due to a load hazard.

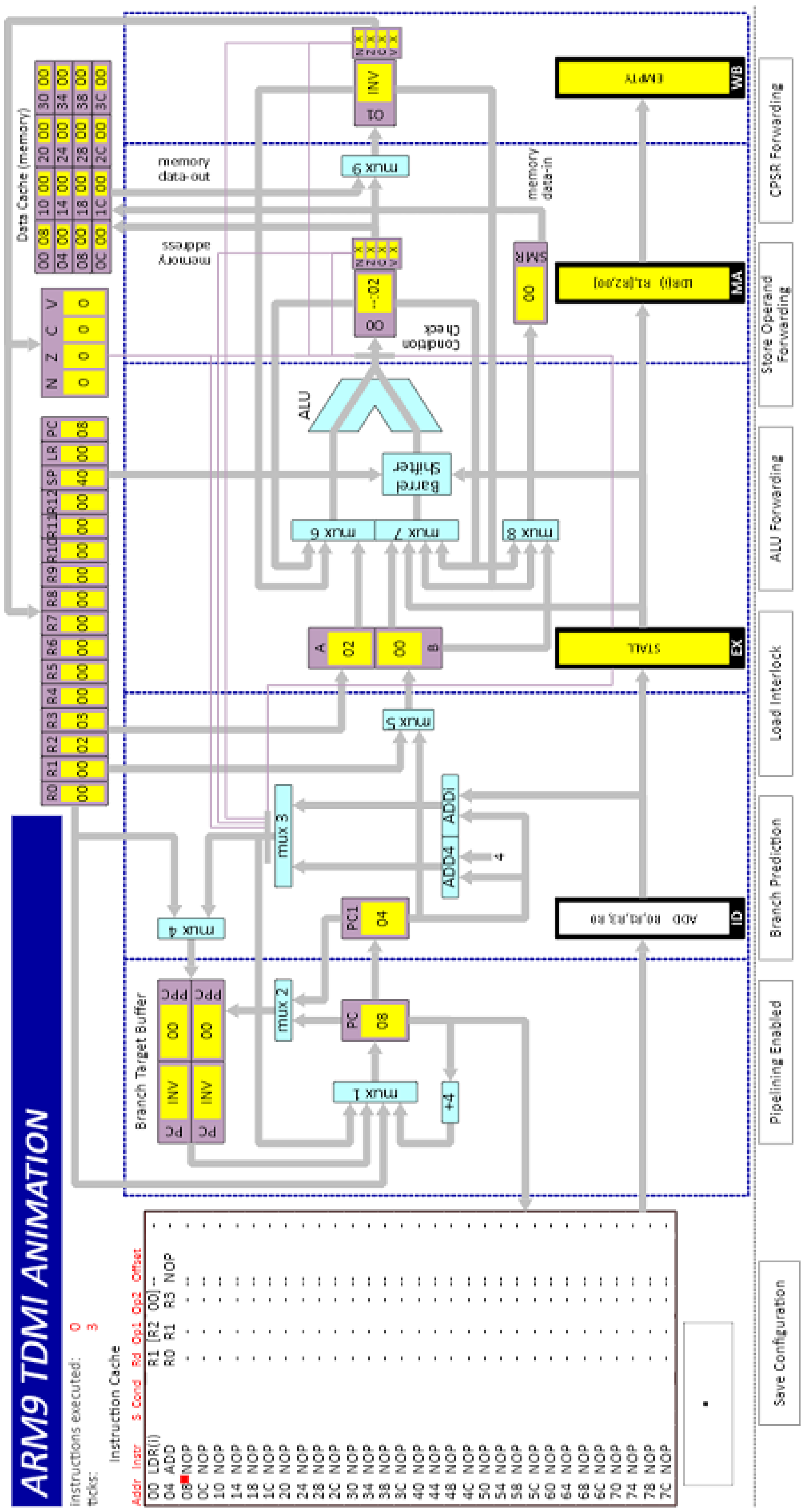


Figure 4.8: ARM9 TDMI Animation: Load Hazard Inducing a Stall

Write Back
The WB stage of the pipeline is responsible for writing values back to the register file and CPSR. This stage demonstrates the second part of the two phase clocking techniques as it shows the register file being written in the first half of the clock cycle (this allows the register file to be read by the ID stage in the second half of the clock cycle preventing the need for further forwarding hardware).

4.1.2 Summary

In summary, the five stage pipeline on the ARM9 TDMI animation has been implemented to demonstrate each of the pipelining concepts and techniques laid out in the Background chapter of this report, the first of the goals outlined in the introduction to the report. The next section will discuss how the ARM Instruction Set and ARM Specific Hardware outlined in the goals of this project are executed and demonstrated respectively.

4.2 Path to Completion

There were a number of stages to converting and building upon the MIPS animation to create the Web-based ARM CPU Animation. This section will explain the implementation of each of these, giving particular attention to the implementation of the ARM conditional execution. As explained in the Design chapter of this report, the Web-based ARM CPU Animation extends the ARM9 TDMI architecture in order to demonstrate all of the techniques desired. Where these extensions have occurred in the implementation of the animation will be outlined in the following subsections.

4.2.1 Learning VivioJS and Understanding the MIPS Animation Code

The first step in building this animation was to learn the VivioJS programming language. This task was more difficult than initially apparent as there was no documentation on the VivioJS language. Moreover, all knowledge of the programming language had to be achieved through trial and error with a limited number of questions answered by Dr. Jeremy Jones who created the language. Next, in order to convert the existing MIPS animation its design and implementation needed to be understood. This itself was a complex task with a steep learning curve as again there was no documentation, furthermore, there was no access to the initial developer Edsko de Vries. This stage on the path to completion of the Web-based ARM CPU

Animation was a serious undertaking and required three weeks of work at the beginning of the development process.

4.2.2 Scaling the Existing Animation

The existing MIPS animation is drawn on a canvas seven hundred and ten pixels wide and four hundred and ninety pixels high giving it a height to width ratio of 0.69. This ratio fit well with computer screens when the animation was designed. However, most modern screens are either 1080p or 4K each of which have a height to width ratio of 0.56, see Equation 1.

$$\frac{1080}{1920}(1080p) = \frac{2160}{3840}(4K) = \frac{9}{16} = 0.5625 \tag{1}$$

As well as this the Web-based ARM CPU Animation is designed to be viewed in a full screen browser window. In most web browsers, such as Google Chrome and Mozilla Firefox, a section at the top of the window contains the address bar with space for tabs and other controls. This gives a height to width ratio of approximately 0.5. For this reason the Web-based ARM CPU Animation was implemented on a canvas two thousand two hundred pixels wide and one thousand and eighty pixels high which fits well in modern browser windows, see Figure 4.9 of the animation running in a full screen Google Chrome browser window on a 1080p laptop screen.

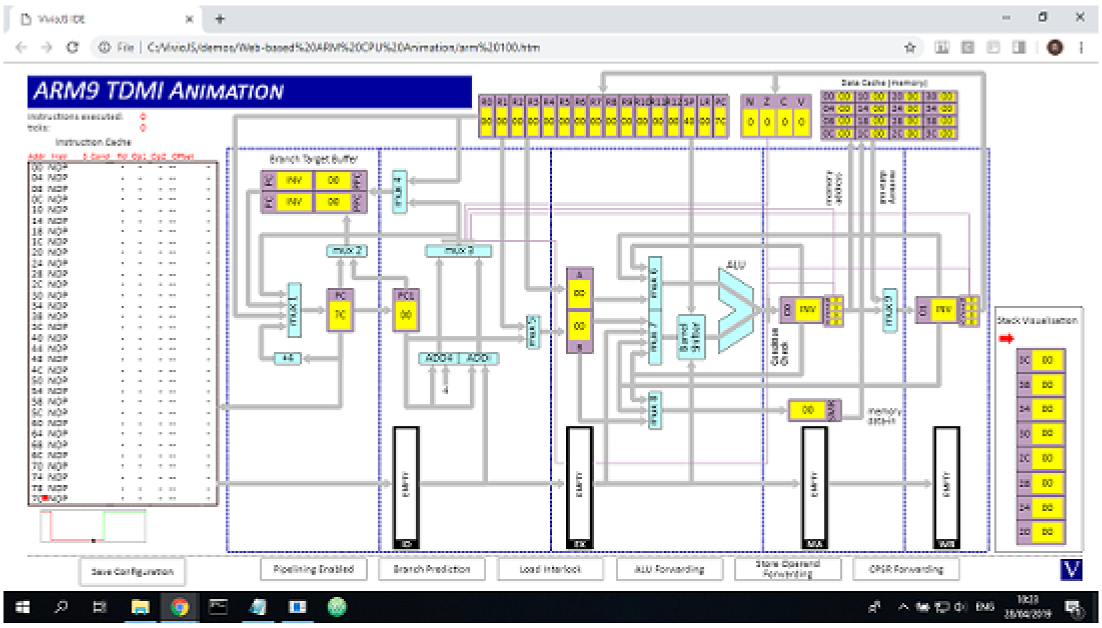


Figure 4.9: Web-based ARM CPU Animation in Full Screen Chrome Browser on 1080p Laptop

The increase in the number of pixels also allows for greater granularity in the positioning of objects in the animation.

To achieve this scaling the python script "scale.py" was written, see Appendix A3 for the full code listing. This takes as input a VivioJS file, the initial canvas size and the desired canvas size. It then compares this file against a series of regular expressions which pick out the coordinates and size of each object and scales them to the new ratio. This was run on the schematic.vin VivioJS file (which contains the positioning and size of all of the animated objects) to scale the existing MIPS animation to our desired screen ratio and granularity.

It should be noted that the HTML5 canvas generated by VivioJS automatically resizes to fit the window it is in but it will not change the height to width ratio of the animation.

4.2.3 Extending the Register File and Data Cache

As explained in the Design chapter, there are only four registers and four data cache memory locations in the existing MIPS animation. The register file and the data cache are both represented as arrays in the MIPS animation. To extend these to show the full sixteen ARM registers and extend the data cache to sixteen places these arrays were first increased in size from four to sixteen and the objects representing each register and location were resized and repositioned to fit cleanly in the top of the animation, see Figures 4.10 and 4.11 below showing the change.

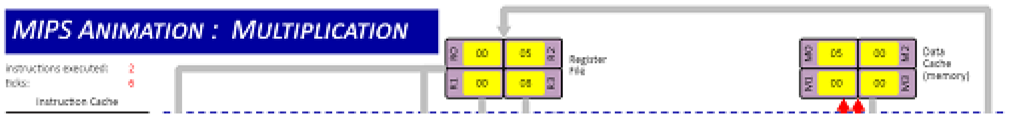


Figure 4.10: MIPS Animation: Register File and Data Cache (Memory) [Jones, 2019]

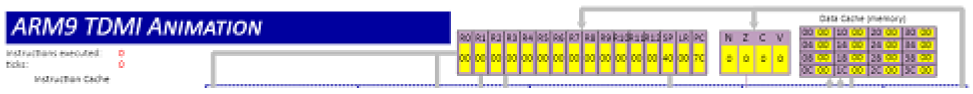


Figure 4.11: ARM9 TDMI Animation: Register File and Data Cache (Memory)

4.2.4 Implementing the Base ARM Instruction Set

To implement the chosen instruction set (as described in the Design chapter of this report), each instruction that can be executed in the pipeline was given a code from zero to fifty. This code maps that instruction to its mnemonic and its operation in the code of the animation. These fifty one instructions (comprising of forty nine user enterable instructions and two system inserted instructions) can be found in Table A1.1 in Appendix 1. When an instruction reaches the execute (EX) stage of the pipeline the function "instrExecute(num instr, string s, num cond, num op1, num op2, num shiftIns, num op3, num n, num z, num c, num v)" is called which takes as arguments the instruction code, the set bit, the condition, each of the operands, the shift instruction and the current state of each of the CPSR flags. Based on these it returns the output of executing that instruction on those operands, see the instructions.vin file in Appendix 2 for a code listing. These instructions are cycled through in the instruction cache by left or right clicking on the instruction on the left hand side.

4.2.5 Extending the Instruction Cache

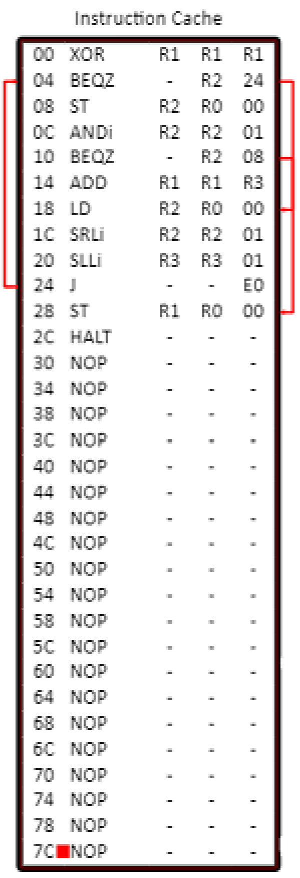
To allow students to enter a comprehensive set of ARM assembly instructions the Instruction Cache (into which students enter the instruction they want to execute) needed to be extended. As explained in the Background chapter of this report the general syntax of an ARM instruction is:

```
<Instruction>{S}{<Condition>} <Destination>,<Source1>,<Source2> {<Shift> <Source3>}
```

and the syntax of the Instructions enterable in the existing MIPS animation is:

```
<Instruction> <Destination>,<Source1>,<Source2>
```

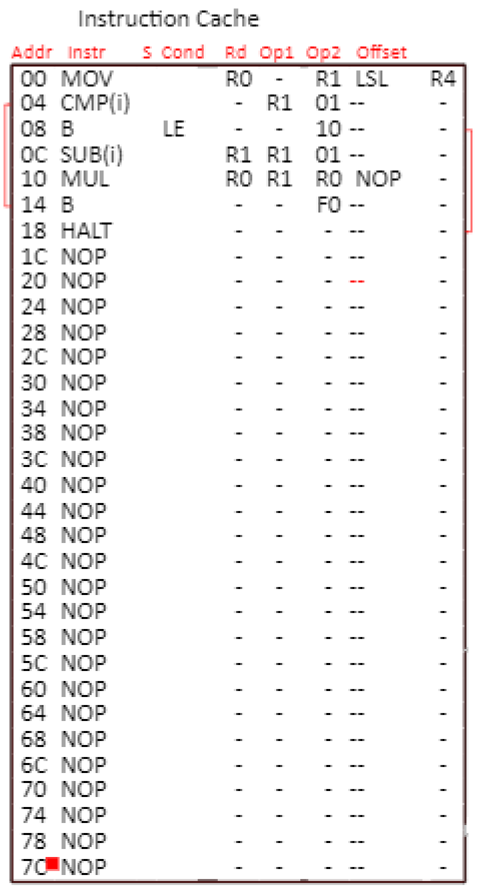
see Figure 4.12 below. Therefore, the Instruction Cache needed to be widened to allow space for the set bit, condition, inline shift instruction and third operand, as shown in Figure 4.13. To achieve this a second Python script ("shift.py") was written to move over everything to the right of the instruction cache. This script takes a VivioJS file, a horizontal position defining a vertical line (everything to the right of this line is moved) and the amount of pixel to move these objects by. Similar to "scale.py" this script was applied to the "schematic.vin" file containing the position of each of the animated objects, creating space for the new wider instruction cache.


Instruction Cache

00	XOR	R1	R1	R1
04	BEQZ	-	R2	24
08	ST	R2	R0	00
0C	ANDi	R2	R2	01
10	BEQZ	-	R2	08
14	ADD	R1	R1	R3
18	LD	R2	R0	00
1C	SRLi	R2	R2	01
20	SLLi	R3	R3	01
24	J	-	-	E0
28	ST	R1	R0	00
2C	HALT	-	-	-
30	NOP	-	-	-
34	NOP	-	-	-
38	NOP	-	-	-
3C	NOP	-	-	-
40	NOP	-	-	-
44	NOP	-	-	-
48	NOP	-	-	-
4C	NOP	-	-	-
50	NOP	-	-	-
54	NOP	-	-	-
58	NOP	-	-	-
5C	NOP	-	-	-
60	NOP	-	-	-
64	NOP	-	-	-
68	NOP	-	-	-
6C	NOP	-	-	-
70	NOP	-	-	-
74	NOP	-	-	-
78	NOP	-	-	-
7C	NOP	-	-	-

Figure 4.12: MIPS Animation: Instruction Cache

[Jones, 2019]


Instruction Cache

Addr	Instr	S	Cond	Rd	Op1	Op2		Offset
00	MOV			R0	-	R1	LSL	R4
04	CMP(i)			-	R1	01	--	-
08	B		LE	-	-	10	--	-
0C	SUB(i)			R1	R1	01	--	-
10	MUL			R0	R1	R0	NOP	-
14	B			-	-	F0	--	-
18	HALT			-	-	-	--	-
1C	NOP			-	-	-	--	-
20	NOP			-	-	-	--	-
24	NOP			-	-	-	--	-
28	NOP			-	-	-	--	-
2C	NOP			-	-	-	--	-
30	NOP			-	-	-	--	-
34	NOP			-	-	-	--	-
38	NOP			-	-	-	--	-
3C	NOP			-	-	-	--	-
40	NOP			-	-	-	--	-
44	NOP			-	-	-	--	-
48	NOP			-	-	-	--	-
4C	NOP			-	-	-	--	-
50	NOP			-	-	-	--	-
54	NOP			-	-	-	--	-
58	NOP			-	-	-	--	-
5C	NOP			-	-	-	--	-
60	NOP			-	-	-	--	-
64	NOP			-	-	-	--	-
68	NOP			-	-	-	--	-
6C	NOP			-	-	-	--	-
70	NOP			-	-	-	--	-
74	NOP			-	-	-	--	-
78	NOP			-	-	-	--	-
7C	NOP			-	-	-	--	-

Figure 4.13: ARM9 TDMI Animation: Instruction Cache

Next, the Instruction class was edited to include local variables to hold the set bit and condition needed for conditional execution as well as the shift instruction and third operand needed for the inline shift operations.

The Instruction Cache is represented by the Instruction Memory class which itself contains an array of thirty two instances of the Instruction class. To achieve the use of the full sixteen ARM registers the instruction cache was altered to allow registers past R0 to R3 to be selected. As is explained in the Design chapter of this report, the operands of an instruction are selected by clicking on the operand to cycle through the possible registers. The MIPS animation implemented this by having the register selected incremented when left clicked and decremented when right clicked, with all increments and decrements taking place modulo four so on left clicking R3 it increments to R0 and so on. This was simply updated to be modulo sixteen for the ARM9 TDMI animation as shown in the code snippet in Figure 4.14 below.

```
when rdt ~> eventMB(num down, num flags, num x, num y) {
    if (!lockCircuit && down && opTypeRdt != OP_TYPE_UNUSED) {
        if (flags & MB_LEFT) {
              vRdt = (vRdt == 15) ? 0 : vRdt + 1;
        } else if (flags & MB_RIGHT)
              vRdt = (vRdt == 0) ? 15 : vRdt - 1;
        initRegs(1);
     }
     return 0;
}
```

Figure 4.14: ARM9 TDMI Animation: Register Selection VivioJS Code

For conditional execution the set bit and condition field were added to the instruction set. The set bit simply required a section in each instruction that when clicked toggles on or off the set bit and displays a capital "S". The condition field was slightly more complicated as the set of conditions needed to be laid out similar to the set of instructions. Each condition was assigned a code that mapped to its name and its operation when being executed. These conditions are cycled through in the same way the instructions are in the instruction cache by left and right clicking.

To allow users to enter the inline shift and rotate instructions (see Table 3.2 for a listing) a second instruction section was added to the Instruction Cache. This operated in the same fashion as the section for entering instruction on the left but only cycled through the shift instructions as these are the only instructions that can be applied inline to operand two of an instruction.

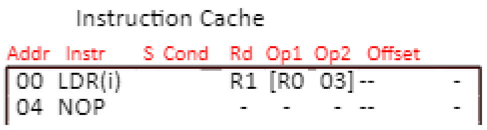
Lastly, in ARM assembly language when entering an address in either a load or store instruction, the operand(s) and shift instruction (if there is one) that amount to the address are entered in square brackets. For example if loading from a base address in R0 offset by three the square brackets appear as shown below:

```
LDR R1,[R0,#3]
```

and if loading from a base address in R0 offset by R3 shifted left by R4 the square brackets appear as shown below:

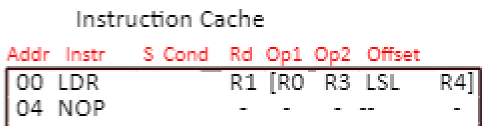
```
LDR R1,[R0,R3,LSL R4]
```

In order to match this convention the position of the square brackets in the Instruction Cache of the animation is dynamic, see Figures 4.15 and 4.16 below.



```
Instruction Cache
Addr  Instr     S Cond   Rd  Op1  Op2  Offset
00  LDR(i)               R1  [R0  03] --     -
04  NOP                  -    -    -  --     -
```

Figure 4.15: ARM9 TDMI Animation: Load with Immediate Offset



```
Instruction Cache
Addr  Instr     S Cond   Rd  Op1  Op2  Offset
00  LDR                  R1  [R0  R3  LSL   R4]
04  NOP                  -    -    -  --     -
```

Figure 4.16: ARM9 TDMI Animation: Load with Register Shifted Offset

4.2.6 Conditional Execution

Conditional execution was the most complicated feature of the ARM instruction set to implement. There were two main reasons for this. Firstly, all instructions in ARM assembly language can be conditionally executed and this requires the CPSR to be checked against every instruction not just branch instructions. Secondly, the CPSR in ARM is part of the register file so it is only written to by instructions during the write back stage of the pipeline. This means there can be data hazards if one instruction sets the CPSR flags and the instruction immediately afterwards checks the CPSR flags.

The first step in implementing conditional execution was to build a class to represent the CPSR. The class defines local variables to represent the current state of each of the CPSR flags as well as the state they will have on the next clock tick. This allows the new values of the CPSR flags calculated during the execution stage to be set without overwriting the values currently in the flags of the CPSR.

The next step was to define a function that given the code for a condition will check this condition against the current CPSR values and return one if the condition is valid and zero otherwise, see the "testCondition(num cond)" function in the "instructions.vin" file in Appendix 2.

After this, three instances of the CPSR class were created. One is the main CPSR and the other two are the forwarding CPSRs to forward the flag values back to the execution (EX) or instruction decode (ID) stages on CPSR data dependencies (one in the memory access (MA) stage and one in the write back (WB) stage), see Figure 4.17 below.

Next, the functionality to set the CPSR flags was written. The function "setFlags" was implemented to set the next values of the CPSR flags. This is called in the "instrExecute" function if the set bit is set.

Lastly, the logic for conditional execution was added. For animation purposes there are two different conditional checks. One check occurs immediately after the EX stage to prevent data processing instructions writing to a forwarding register if conditionally not executed, see the black bar marked "Condition Check" in Figure 4.17 below which is showing an add instruction being prevented from writing to the first forwarding register "O0". The other check occurs in the ID stage which prevents branch instructions from being executed when their conditional field is not valid with the current CPSR flags, see the bar above "Mux 3" in Figure 4.17 below.

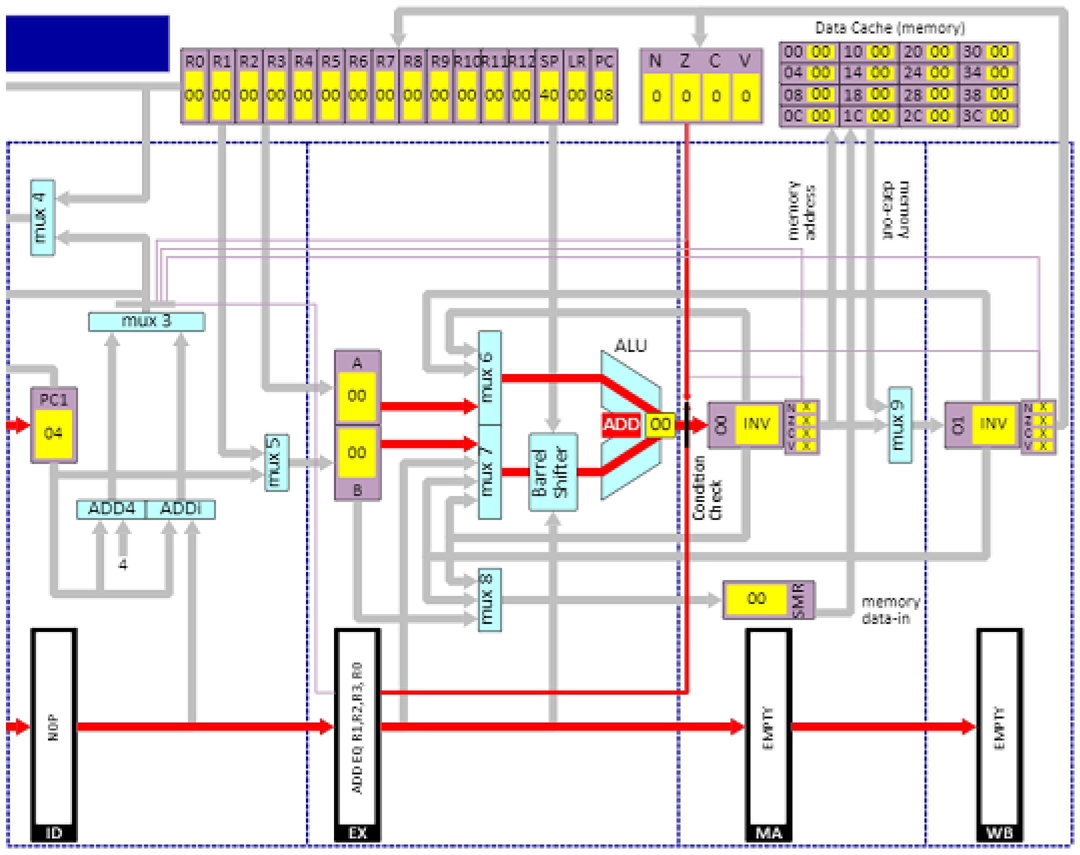


Figure 4.17: ARM9 TDMI Animation: CPSR and Forwarding CPSRs

4.2.7 Inline Shifting

The inline shift operations executable in ARM CPUs are one of the most powerful features as they can be executed in a single clock cycle. To demonstrate this the Web-based ARM CPU Animation visualises the Barrel Shifter on the operand two input to the ALU, see an example of the EX stage of the pipeline executing a load with a shifted offset in Figure 4.18 on the following page. This shift of operand two is performed at the beginning of the "instrExecute" function, performing the shift operation on operand two before the main instruction is executed, see the code snippet performing this shift in Figure 4.19 on the following page.

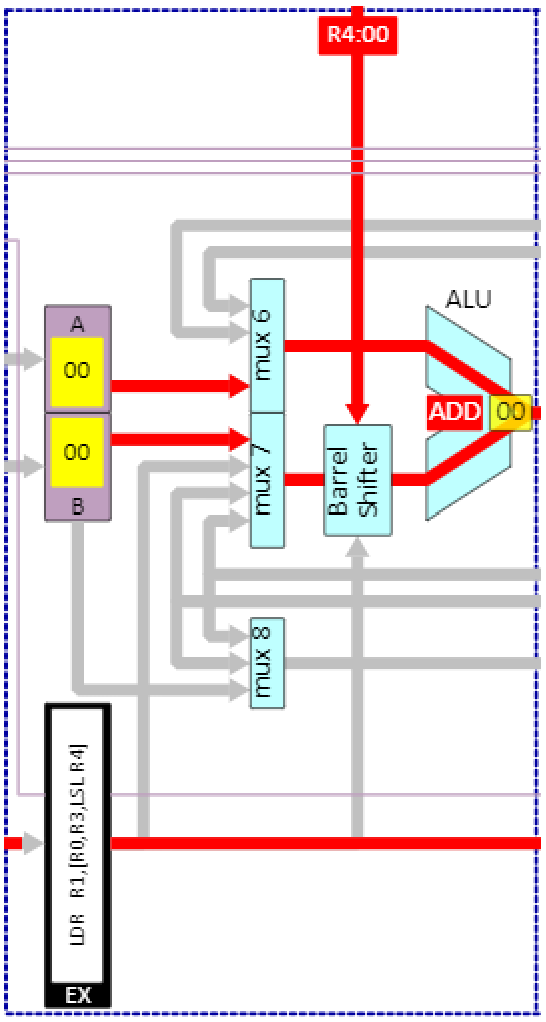


Figure 4.18: ARM9 TDMI Animation: Barrel Shifter

```
//Perform Shift on OP2
if(shiftIns == LSL || shiftIns == LSLi){
    op2 = (op2 << op3) & 0xFF;
} else if(shiftIns == LSR || shiftIns == LSRi){
    op2 = (op2 >> op3) & 0xFF;
} else if(shiftIns == ASR || shiftIns == ASRi){
    num sign = op2 & 0x80;
    op2 = (op2 >> op3) & 0xFF;
    if(sign){
         op2 = (op2 | (0xFF << (8-op3))) & 0xFF;
    }
} else if(shiftIns == ROR || shiftIns == RORi){
     op3 = op3%8;
     op2 = ((op2 >> op3) | (op2 << (8-op3))) & 0xFF;
}
```

Figure 4.19: ARM9 TDMI Animation: Register Selection VivioJS Code

4.2.8 Stack Animation

The stack animation is simply a different view of the top eight locations in the Data Cache (Memory), drawn vertically on the bottom right of the animation. The stack pointer is implemented as eight different arrows each drawn pointing to one of these memory locations. The arrow corresponding to the value in the Stack Pointer register (SP or R14) is shown and all others are hidden. See Figure 3.6 in the Design Chapter of this report.

4.2.9 Summary

In summary, the Web-based ARM CPU Animation was implemented such that it achieves our other two goals. It executes user defined ARM assembly language programs and demonstrates ARM specific hardware (including the CPSR and Barrel Shifter).

5 Evaluation

This chapter evaluates the Web-based ARM CPU Animation against the best practices highlighted in the Background chapter of this report and against the existing MIPS animation.

5.1 Evaluation Against Best Practices

In "Exploring the role of visualization and engagement in computer science education" [Fouh et al., 2012] suggest a set of best practices for creating useful educational animations for computer science. As shown in Table 5.1 on the following page, the Web-based ARM CPU Animation follows nine of these eleven best practices and is designed such that the remaining two can easily be added. The animation shows a help screen on start up to provide a resource that helps learners interpret the animation. It provides sample programs and accepts custom programs adapting to the knowledge of the learner. It shows the clock ticks and instructions executed to show performance information. It shows previously executed instructions to include execution history. It can be run forwards and backwards, supporting flexible execution control. It is configurable, supporting learner built visualisations. It allows users to write and execute their own programs supporting custom input. It is complemented by the explanations in the CS3021: Computer Architecture II Course. Providing dynamic questions and feedback was not within the scope of the project.

Table 5.1: Assessment of the Web-based ARM CPU Animation Against Best Practices for Educational Animations in Computer Science

Practice	**Achieved by Animation**	**Note**
Provide resources that help learners interpret the graphical representation	Yes	Help screen on start up
Adapt to the knowledge level of the user	Yes	Sample programs for students starting out
Provide multiple views	Yes	Stack is a different view of memory
Include performance information	Yes	Clock ticks and instructions executed displayed
Include execution history	Yes	Instructions already run shown
Support flexible execution control	Yes	Animation can be run forwards, backwards and in steps
Support learner-built visualizations	Yes	Animation features can be toggled on or off
Support custom input data sets	Yes	User can enter their own programs
Support dynamic questions	Not Yet	No pop up questions
Support dynamic feedback	Not Yet	No feedback on user input
Complement visualizations with explanations	Yes	Animation tied to CS3021 Course

5.2 Evaluation Against Existing MIPS animations

In this section I will assess the Web-based ARM CPU Animation in the context of the existing MIPS animation. The Web-based ARM CPU Animation was built upon the existing animation, therefore, it covers all of the same functionality as the original. It also confers more educational benefit to its users as it demonstrates ARM specific hardware giving students a deeper understanding of one of the most widely used CPU architectures. Furthermore, the Web-based ARM CPU Animation supports the execution of fifty user enterable instructions, over fifty percent more than the thirty-two supported in the existing animation. The twenty-three ARM instructions operating on registers only are extended with a set bit, nineteen different condition codes and eight different shift instructions. The twenty-two ARM instructions operating on registers and immediate values are extended with a set bit and nineteen different condition codes. This gives users over seven thousand unique instructions than can be executed in the animated pipeline. The Web-based ARM CPU includes a larger register file and data cache allowing users to write more complex programs in the instruction cache. Lastly, the Web-based ARM CPU Animation includes a full descending stack animation, demonstrating to students another core concept of ARM Assembly Language programming.

6 Conclusion

In conclusion, the Web-based ARM CPU Animation has achieved all of the goals set out at the start of this project. A synopsis of the current state of the Web-based ARM CPU is explained below, followed by a few features that that could be added to the animation in future and a summary of the achievements of the project.

6.1 Synopsis

The Web-based ARM CPU Animation is an animation of the ARM9 TDMI CPU pipeline with extra hardware to demonstrate the pipelining techniques not available on the ARM9. It demonstrates each of the concepts and techniques taught to students in introductory pipelining courses such as CS3021: Computer Architecture II at Trinity College Dublin. It achieves this through executing user written ARM assembly language programs and demonstrating ARM specific hardware.

6.2 Future Work

There are a few features that (although outside the scope of the project) would be beneficial if implemented.

Firstly, an updated method of entering the user defined programs could be added. This could be achieved by implementing menus, which are soon to be a feature of the VivioJS language. This could also be achieved by developing an ARM assembly language text editor that runs in browser in place of the instruction cache. This could type check user programs and parse them into instructions to be executed by the pipeline.

Secondly, dynamic questions and feedback could be included, in line with the best practices outlined in the Evaluation chapter of this report. Perhaps this could be a future final year project.

Lastly, an organised study into the educational benefits of the animation Web-based ARM CPU Animation could be conducted as it is ready to be tested by students studying pipelining.

6.3 Summary

The Web-based ARM CPU Animation has succeeded in achieving all three of the goals laid out in the introduction to this report. It is an animation of an ARM Pipeline that:

- Demonstrates each of the pipelining concepts and techniques in the CS3021 (or any) computer architecture course
- Executes ARM Assembly Language Instructions
- Demonstrates ARM Specific Hardware

Additionally, it supports features that were not initially included in the scope of this project, including conditional execution and a stack animation. All of this is packaged in a HTML5 and JavaScript based animation that can be hosted on any web page and run in browser making it extremely portable and easy to access. The Web-based ARM CPU Animation has surpassed the state of the art in educational pipelining animations and is ready to be used by computer science students and instructors around the world.

References

Eric Fouh, Monika Akbar, and Clifford A. Shaffer. The Role of Visualization in Computer Science Education. *Computers in the Schools*, 29(1-2):95–117, 2012. doi: 10.1080/07380569.2012.651422. URL `https://doi.org/10.1080/07380569.2012.651422`.

S. Furber. The ARM10TDMI pipeline. Reproduced as Figure 9.9 in [Furber, 2000c], 2000a.

S. Furber. ARM7TDMI and ARM9TDMI pipeline comparison. Reproduced as Figure 9.7 in [Furber, 2000c], 2000b.

S. Furber. *ARM system-on-chip architecture*. Pearson Education, 2000c.

Gem5. Gem5 Visualization, 2016. URL `http://www.m5sim.org/Visualization`. Accessed 20/04/2019.

CD Hundhausen, SA Douglas, and JTA Statsko. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages & Computing*, 2002.

J. Jones. Vivio - A System for Creating Interactive Reversible E-Learning Animations for the WWW. In *2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 131–133, Sep. 2004. doi: 10.1109/VLHCC.2004.63.

J Jones. VivioJS Animations, 2019. URL `https://www.scss.tcd.ie/Jeremy.Jones/VivioJS/vivio.htm`. Accessed 20/04/2019.

Thomas Naps, Stephen Cooper, Boris Koldehofe, Charles Leska, Guido Rö, Wanda Dann, Ari Korhonen, Lauri Malmi, Jarmo Rantakokko, Rockford J. Ross, Jay Anderson, Rudolf Fleischer, Marja Kuittinen, and Myles McNally. Evaluating the Educational Impact of Visualization. *SIGCSE Bull.*, 35(4):124–136, June 2003. ISSN 0097-8418. doi: 10.1145/960492.960540. URL `http://doi.acm.org/10.1145/960492.960540`.

Thomas L. Naps, Guido Rö, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and J. Ángel Velázquez-Iturbide. Exploring the Role of Visualization and Engagement in Computer Science Education. *SIGCSE Bull.*, 35(2):131–152, June 2002. ISSN 0097-8418. doi: 10.1145/782941.782998. URL `http://doi.acm.org/10.1145/782941.782998`.

Morten Borup Petersen. Ripes, 2019. URL `https://github.com/mortbopet/Ripes`. Accessed 20/04/2019.

SCSS at Trinity College Dublin. BA (Mod) in Computer Science Master in Computer Science (MCS), 2017a. Accessed 20/04/2019.

SCSS at Trinity College Dublin. CS3021: Computer Architecture II Module Descriptor, 2017b. URL `https://scss.tcd.ie/modules/?m=CS3021`. Accessed 20/04/2019.

A1 Instructions and Conditions

A1.1 ARM Instruction Set

Table A1.1: The Standard Instructions Implemented

#	Mnemonic	Interpretation	Note
1	*NOP*	Nothing	For Animation Purposes
2	ADD	Add	
3	ADC	Add with Carry	
4	SUB	Subtract	
5	SBC	Subtract with Carry	
6	RSB	Reverse Subtract	
7	RSC	Reverse Subtract with Carry	
8	MUL	Multiply	
9	AND	Logical AND	
10	ORR	Logical OR	
11	EOR	Exclusive OR	
12	BIC	Bit Clear	
13	LSL	Logical Shift Left	
14	LSR	Logical Shift Right	
15	ASR	Arithmetic Shift Right	
16	ROR	Rotate Right	
17	ADDi	Add (Immediate)	
18	ADCi	Add with Carry (Immediate)	
19	SUBi	Subtract (Immediate)	
20	SBCi	Subtract with Carry (Immediate)	
21	RSBi	Reverse Subtract (Immediate)	
22	RSCi	Reverse Subtract with Carry (Immediate)	
23	ANDi	Logical AND (Immediate)	
24	ORRi	Logical OR (Immediate)	

#	Mnemonic	Interpretation	Note
25	EORi	Exclusive OR (Immediate)	
26	BICi	Bit Clear (Immediate)	
27	LSLi	Logical Shift Left (Immediate)	
28	LSRi	Logical Shift Right (Immediate)	
29	ASRi	Arithmetic Shift Right (Immediate)	
30	RORi	Rotate Right (Immediate)	
31	CMN	Compare Negative	
32	CMP	Compare	
33	TEQ	Test bitwise equality	
34	TST	Test bits	
35	CMNi	Compare Negative (Immediate)	
36	CMPi	Compare (Immediate)	
37	TEQi	Test bitwise equality (Immediate)	
38	TSTi	Test bits (Immediate)	
39	MOV	Move register or constant	
40	MVN	Move negative register	
41	LDR	Load register from memory	
42	STR	Store register to memory	
43	MOVi	Move register or constant (Immediate)	
44	MVNi	Move negative register (Immediate)	
45	LDRi	Load register from memory (Immediate)	
46	STRi	Store register to memory (Immediate)	
47	B	Branch	
48	BL	Branch with Link	
49	*HALT*	Stop running	For Animation Purposes
50	*STALL*	To visualise pipeline stalls	Not Enterable
51	*EMPTY*	To visualise empty pipeline	Not Enterable

NB: Mnemonics shown in italics do not correspond to ARM assembly instructions, they were implemented for animation purposes.

A1.2 ARM Conditional Execution Codes

Table A1.2: The Conditions Implemented

#	Mnemonic	Interpretation	Valid Case
1		equivalent to AL	Default
2	EQ	Equal	Z set
3	NE	Not Equal	Z clear
4	HS	Unsigned Higher or same	C set
5	CS	C set	C set
6	LO	Unsigned Lower or Same	C clear
7	CC	C clear	C clear
8	MI	Negative	N set
9	PL	Positive or Zero	N clear
10	VS	Overflow	V set
11	VC	No Overflow	V clear
12	HI	Unsigned Higher	C set and Z clear
13	LS	Unsigned Lower or Same	C clear or Z set
14	GE	Greater Than or Equal	N set and V set, or N clear and V clear
15	LT	Less Than	N set and V clear, or N clear and V set
16	GT	Greater Than	Z clear, and either N set and V set, or N clear and V clear
17	LE	Less Than or Equal	Z set, or N set and V clear, or N clear and V set
18	AL	Always	
19	NV	Never	

A2 VivioJS Code

```
//
// arm.viv
//
// Simulation of the ARM
// Written by Rónán Dowling-Cullen, Final Year Computer Science 2018/19DLX
// Building on the simulation of the DLX written by Edsko de Vries, Summer 2003


#include "defaults.vin"
#include "config.vin"
#include "instructions.vin"
#include "instructionmemory.vin"
#include "instructionregister.vin"
#include "register.vin"
#include "component.vin"
#include "alu.vin"
#include "animatedpipe.vin"
#include "animatedclock.vin"
#include "button.vin"
#include "cpsr.vin"
#include "stackpointer.vin"
#include "schematic.vin"
#include "execution.vin"
#include "buttonhandlers.vin"

// eof
```

```
//
// animatedclock.vin
//
// Simulation of the ARM
// Written by Rónán Dowling-Cullen, Final Year Computer Science 2018/19DLX
// Building on the simulation of the DLX written by Edsko de Vries, Summer 2003

Pen redClockPen = SolidPen(SOLID, 1, RED, ROUND_START | ROUND_JOIN | ROUND_END);
Pen greenClockPen = SolidPen(SOLID, 1, GREEN, ROUND_START | ROUND_JOIN | ROUND_END);
Pen orangeClockPen = SolidPen(SOLID, 1, ORANGE, ROUND_START | ROUND_JOIN | ROUND_END);

//
// AnimatedClock
//
class AnimatedClock(num x, num y, num w, num h) extends Group(0, 0, x, y, 0, 0, w, h) {

num cw = w;
num chw = cw / 2;
num ch = h - 6;

num stall = 0, type = 0;

           setClipPath(R$(0, 0, w, h));


Rectangle clkDisplay = Rectangle2(0, 0, blackPen, whiteBrush, 0, 0, w, h);

           clkDisplay.setRounded(2, 2);


Line prev_clock = Line(activePipesLayer, 0, redClockPen, -chw+chw/5,3+ch, 0,0, 0,-ch, chw,0, 0,ch, chw,0);
Line next_clock = Line(activePipesLayer, 0, greenClockPen, chw+chw/5,3+ch, 0,0, 0,-ch, chw,0, 0,ch, chw,0);

Rectangle dot = Rectangle2(activePipesLayer, 0, 0, blackBrush, w/2-3, h-6, 6, 6);


num canUpdate;

function clockCycle(num length) {


num l2 = length / 2;
num l5 = length / 5;
num l10 = length / 10;

canUpdate = 0;

prev_clock.translate(-chw, 0, l2, 1, 0);
next_clock.translate(-chw, 0, l2, 1, 0);
dot.translate(0, -ch, l5, 1, 0);

wait(l2);

prev_clock.translate(-chw, 0, l2, 1, 0);
next_clock.translate(-chw, 0, l2, 1, 0);
dot.translate(0, ch, l5, 1, 0);

wait(l10);
canUpdate = 1;
prev_clock.translate(2*cw, 0);
prev_clock.setPen(stall ? (type ? orangeClockPen : redClockPen) : greenClockPen)

wait(l5*2);

Line t = next_clock;
next_clock = prev_clock;
prev_clock = t;

if (stall)
stall--;

}

//
// Indicate that the next s clock cycles are extra ("stalled") cycles
//
function setStall(num s, num t) {
stall = s;
type = t;
if (canUpdate)
prev_clock.setPen(stall ? (type ? orangeClockPen : redClockPen) : greenClockPen);

}
```

```
}

// eof

//
// animatedpipe.vin
//
// Simulation of the ARM
// Written by Rónán Dowling-Cullen, Final Year Computer Science 2018/19DLX
// Building on the simulation of the DLX written by Edsko de Vries, Summer 2003

class AnimPipe() {

num w = 10;                                 // line width
                                 // could be static / global
           num n = 0;                       // # points
           num px[0];                       // x co-ordinates
           num py[0];                       // y co-ordinates
           num ls[0];                       // line segment lengths
num ll = 0;                                 // line length (assumes a Manhattan layout)
           num head = 1;                    // if 1 draw line with an arrow60 end

Pen bgPen0 = SolidPen(SOLID, w, GRAY192, BEVEL_JOIN | BUTT_END);            // could be static / global
           Pen bgPen1 = SolidPen(SOLID, w, GRAY192, BEVEL_JOIN | ARROW60_END);                    // could be static / global

Pen fgPen0 = SolidPen(SOLID, w, RED, BEVEL_JOIN | BUTT_END);                           // could be static / global
Pen fgPen1 = SolidPen(SOLID, w, RED, BEVEL_JOIN | ARROW60_END);                        // could be static / global

Line bgLine = Line(inactivePipesLayer, 0, bgPen1, 0, 0);
           Line fgLine = Line(activePipesLayer, 0, fgPen0, 0, 0);

function setOpacity(num opacity) {
bgLine.setOpacity(opacity);
fgLine.setOpacity(opacity);
}

function setHead(num h) {
head = h ? 1 : 0;
                      bgLine.setPen(head ? bgPen1 : bgPen0);
                      fgLine.setPen(fgPen0);
}

function addPoint(num x, num y) {
px[n] = x;
py[n] = y;
bgLine.setPt(n, x, y);
n++;
}

function calcLength() {
num dx, dy;
ll = 0;
for (num i = 0; i < n - 1; i++) {
dx = px[i + 1] - px[i];
dy = py[i + 1] - py[i];
ll += ls[i] = sqrt(dx*dx + dy*dy);
}
}

// Only call on inactive arrows
function setPoint(num n, num x, num y) {
px[n] = x;
py[n] = y;
bgLine.setPt(n, x, y);
}

function reset() {
                      fgLine.setNPts(0);
fgLine.setPen(fgPen0);
}

function animate(num steps) {
                      num l = 0, s, ss = 0;
                      num d = 0;
                      calcLength();
                      fgLine.setPt(0, px[0], py[0]);
                      fgLine.setPen(fgPen0);
                      for (num i = 1; i < n; i++) {
                                 fgLine.setPt(i, px[i - 1], py[i - 1]);
                                 l += ls[i - 1];
                                 s = round(l * steps / ll);
                                 fgLine.setPt(i, px[i], py[i], s - ss, 1, 1);
```

```
                                 ss = s;
                      }
                      if (head)
                                 fgLine.setPen(fgPen1);
           }
}

// eof

//
// button.vin
//
// Simulation of the ARM
// Written by Rónán Dowling-Cullen, Final Year Computer Science 2018/19DLX
// Building on the simulation of the DLX written by Edsko de Vries, Summer 2003

Brush buttonBrush = SolidBrush(WHITE);
Brush hoverBrush = SolidBrush(GRAY224);

class Button(num x, num y, num w, num h, string caption, num ID) {

Rectangle label = Rectangle2(0, 0, blackPen, buttonBrush, x, y, w, h, blackPen, font, caption);

when label ~> eventEE(num enter, num x, num y) {
                      label.setBrush(enter ? hoverBrush : buttonBrush);
                      return 0;
}

function setCaption(string caption) {
label.setTxt(caption);
}

function showLocked(num locked) {
label.setFont(locked ? fontST : font);
}

}

// eof


//
// buttonhandlers.vin
//
// Simulation of the ARM
// Written by Rónán Dowling-Cullen, Final Year Computer Science 2018/19DLX
// Building on the simulation of the DLX written by Edsko de Vries, Summer 2003

function setlocked() { // joj
           num b_locked = locked || lockCircuit; // joj
           buttonPE.showLocked(b_locked); // joj
           buttonBP.showLocked(b_locked); // joj
           buttonLI.showLocked(b_locked); // joj
           buttonAF.showLocked(b_locked); // joj
           buttonSF.showLocked(b_locked); // joj
           buttonZF.showLocked(b_locked); // joj
} // joj

//
// run animation
//
// run until HALT instruction executed
//
function run() {
           wait(1);
           locked = 1;
           setlocked();
           while(1) {

                      fork(animClock.clockCycle((peMode == PIPELINING_ENABLED) ? 80 : 400));
                      exec();
                      if (((regWB.vIns == HALT) && (peMode == PIPELINING_ENABLED)) ||
                                 ((regID.vIns == HALT) && (peMode == PIPELINING_DISABLED))) {
                                 stop();
                                 if (haltOnHalt)

                                            break;
                      }
                      wait(1);
           }
}

//
```

```
// pipeline enable
//
when buttonPE.label ~> eventMB(num down, num flags, num x, num y) {
          if (down && (flags & MB_LEFT) && (!locked) && (!lockCircuit)) {
                    setPEMode((peMode + 1)% 2);
                    resetCircuit();
          }
          return 0;
}

//
// branch prediction
//
when buttonBP.label ~> eventMB(num down, num flags, num x, num y) {
          if (down && (flags & MB_LEFT) && (!locked) && (!lockCircuit)) {
                    setBPMode((bpMode + 1) % 3);
                    resetCircuit();
          }
          return 0;
}

//
// load interlocked
//
when buttonLI.label ~> eventMB(num down, num flags, num x, num y) {
          if (down && (flags & MB_LEFT) && (!locked) && (!lockCircuit)) {
                    setLIMode((liMode + 1) % 2);
                    resetCircuit();
          }
          return 0;
}

//
// ALU forwarding
//
when buttonAF.label ~> eventMB(num down, num flags, num x, num y) {
          if (down && (flags & MB_LEFT) && (!locked) && (!lockCircuit)) {
                    setAFMode((afMode + 1) % 3);
                    resetCircuit();
          }
          return 0;
}

//
// store forwarding
//
when buttonSF.label ~> eventMB(num down, num flags, num, num) {
          if (down && (flags & MB_LEFT) && (!locked) && (!lockCircuit)) {
                    setSFMode((sfMode + 1) % 3);
                    resetCircuit();
          }
          return 0;
}

//
// handle CPSR button press
//
when buttonZF.label ~> eventMB(num down, num flags, num, num) {
          return 0;
}

//
// save configuration
//
// NB: generating unnecessary quotes for compatibilty with previous versions
//
when buttonSP.label ~> eventMB(num down, num flags, num x, num y) {
          if (down && (flags & MB_LEFT)) {


                    num lp1, opcode, reg;
                    Instruction instr;

string s = "saveanim.php?state=";

for (lp1 = 0; lp1 < 32; lp1++) {
instr = im.instruction[lp1];
opcode = (instr.vIns << 24) | (instr.vRdt << 16) | (instr.vRs1 << 8) | (instr.vRs2);
s = sprintf("%si%d='0x%08X' ", s, lp1, opcode);
}

for (lp1 = 0; lp1 < 4; lp1++) {
reg = regFile[lp1].value;
s = sprintf("%sr%d='0x%02X' ", s, lp1, reg);
}

for (lp1 = 0; lp1 < 4; lp1++) {
reg = memory[lp1].value;
s = sprintf("%sm%d='0x%02X' ", s, lp1, reg);
}

s = sprintf("%speMode='%d' bpMode='%d' liMode='%d' afMode='%d' sfMode='%d' zfMode='%d'", s, peMode, bpMode, liMode, afMode, sfMode,
zfMode);
                    getURL(s);

          }
          return 0;
}

//
// Vivio logo
//
when logo ~> eventMB(num down, num flags, num x, num y) {
if (down && (flags & MB_LEFT))
getURL("https://www.scss.tcd.ie/Jeremy.Jones/VivioJS/vivio.htm");
return 0;
}

//
// load program from database
//
when title ~> eventMB(num down, num flags, num, num) {
if (down && (flags & MB_LEFT))
getURL("showanim.php");
}

//
// change initial builtin program and reset
//
when imLabel ~> eventEE(num enter, num x, num y) {
          imLabel.setBrush(enter? gray192Brush : whiteBrush);
          imLabel.setTxtPen(enter ? redPen : blackPen);
          return 0;
}

when imLabel ~> eventMB(num down, num flags, num x, num y) {
          if (down && (flags & MB_LEFT)) {
                    example = (example == maxexample) ? 0 : example + 1;
                    setArg("example", example.toString());
                    reset();
          }
          return 0;
}

run();


// eof



//
// component.vin
//
// Simulation of the ARM
// Written by Rónán Dowling-Cullen, Final Year Computer Science 2018/19DLX
// Building on the simulation of the DLX written by Edsko de Vries, Summer 2003

Brush componentBrush = SolidBrush(LIGHT_BLUE);
Font componentFont = Font("Calibri", 24);

class Component(num _x, num _y, num _w, num _h, string caption) {

          num x = _x;
          num y = _y;
          num w = _w;
          num h = _h;

Rectangle bg = Rectangle2(activePipesLayer, 0, blackPen, componentBrush, x, y, w, h);
          bg.setRounded(2, 2);
Rectangle label;

if (w >= h) {
label = Rectangle2(activePipesLayer, 0, 0, 0, x, y, w, h, 0, componentFont, caption);
} else {
label = Rectangle(activePipesLayer, 0, 0, 0, x + w/2 - 3, y + h/2, -w/2, -h/2, w, h, 0, componentFont, caption);
                    label.rotate(-90);
```

```
}

function setOpacity(num opacity) {
bg.setOpacity(opacity);
label.setOpacity(opacity);
}

}

// eof

//
// config.vin
//
// Simulation of the ARM
// Written by Rónán Dowling-Cullen, Final Year Computer Science 2018/19DLX
// Building on the simulation of the DLX written by Edsko de Vries, Summer 2003

//Run Test on register output of every instruction
const num testOps = 1;

const num WIDTH = 2200;
const num HEIGHT = 1080;

const num maxexample = 10;
num example = 0;                                                                                                    // must be 0

setViewport(0, 0, WIDTH, HEIGHT, 1);

setBgBrush(whiteBrush);

Font sFont = Font("Calibri", 20);
Font font = Font("Calibri", 22);
Font fontST = Font("Calibri", 22, STRIKETHROUGH);

// Layers
Layer valuesLayer = Layer(1, 3);
Layer inactivePipesLayer = Layer(2, 3);
Layer activePipesLayer = Layer(3, 0);
Layer helpLayer = Layer(5, 0);

Brush bhighlight = SolidBrush(RED);

//
// Schematic configuration
//
num lockCircuit = 0;

//
// Some constant definitions
//
const num NO_STALL = 0;
const num DATA_STALL = 1;
const num CTRL_STALL = 2;
const num MUL_STALL = 3;

const num PIPELINING_ENABLED = 0;
const num PIPELINING_DISABLED = 1;

const num BRANCH_PREDICTION = 0;
const num BRANCH_INTERLOCK = 1;
const num DELAYED_BRANCHES = 2;

const num LOAD_INTERLOCK = 0;
const num NO_LOAD_INTERLOCK = 1;

const num ALU_FORWARDING = 0;
const num ALU_INTERLOCK = 1;
const num NO_ALU_INTERLOCK = 2;

const num FORWARDING_TO_SMDR = 0;
const num STORE_INTERLOCK = 1;
const num NO_STORE_INTERLOCK = 2;

const num ZERO_FORWARDING = 0;
const num ZERO_INTERLOCK = 1;
const num NO_ZERO_INTERLOCK = 2;

//
// Global variables
//
num stall = NO_STALL; // NO_STALL/DATA_STALL (leave the PC)/CTRL_STALL (update the PC)
num btbLast = 1; // last accessed entry in the BTB
num updateBTB = 0; // update the BTB on the next clock cycle

num peMode = 0; // Pipeline enabled
num bpMode = 0; // Branch prediction/branch interlock/delayed branches
num liMode = 0; // Load interlock/No load interlock
num afMode = 0; // ALU Forwarding/ALU Interlock/No ALU Interlock
num sfMode = 0; // Forwarding to SMDR/Store Interlock/No Store Interlock
num zfMode = 0; // Zero Forwarding/Zero Interlock/No Zero Interlock

//
// Running state
//
num locked = 0; //
num instrCount = 0; //
num tickCount = 0; //

// eof

//
// cpsr.vin
//
// Simulation of the ARM
// Written by Rónán Dowling-Cullen, Final Year Computer Science 2018/19DLX
// Building on the simulation of the DLX written by Edsko de Vries, Summer 2003

Brush cpsrBrush = SolidBrush(PURPLE);
Brush cpsrValueBrush = SolidBrush(WHITE);
Font lFont = componentFont;
Font vFont = font;

class CPSR(num x, num y, num w, num h, num forwarding) {

num n = 0;
        num z = 0;
        num c = 0;
        num v = 0;

        num nN = 0;
        num nZ = 0;
        num nC = 0;
        num nV = 0;

        num invalid = 0;

        //Co-ordinates of n,z,c and v parts
        num nx = x;
        num zx = x + w/4;
        num cx = x + 2*(w/4);
        num vx = x + 3*(w/4);

        num ny = y + h/3;
        num zy = y + h/3;
        num cy = y + h/3;
        num vy = y + h/3;

        if(forwarding) { //Small so need to change font sizes
                lFont = Font("Calibri", 16);
                vFont = Font("Calibri", 16);
        }

        // Outer Rectangle
        Rectangle outer;

        // Labels
        Rectangle nLabel;
        Rectangle zLabel;
        Rectangle cLabel;
        Rectangle vLabel;

        // Values
Rectangle nValue;
Rectangle zValue;
Rectangle cValue;
Rectangle vValue;


        if(h > w) {

                //Co-ordinates of n,z,c and v parts
                nx = x + w/3;
                zx = x + w/3;
                cx = x + w/3;
                vx = x + w/3;
                ny = y;
                zy = y + h/4;
                cy = y + 2*(h/4);
                vy = y + 3*(h/4);

                // Outer Rectangle
                outer = Rectangle2(0, 0, blackPen, cpsrBrush, x, y, w, h+4);                //Draw outer rectangle

                // Labels
                nLabel = Rectangle(0, 0, 0, 0, x+2, ny, 0, 0, w/4, h/3, 0, lFont, "N");
                zLabel = Rectangle(0, 0, 0, 0, x+2, zy, 0, 0, w/4, h/3, 0, lFont, "Z");
                cLabel = Rectangle(0, 0, 0, 0, x+2, cy, 0, 0, w/4, h/3, 0, lFont, "C");
                vLabel = Rectangle(0, 0, 0, 0, x+2, vy, 0, 0, w/4, h/3, 0, lFont, "V");

                // Values
        nValue = Rectangle2(valuesLayer, 0, 0, yellowBrush, nx, ny+4, 2*w/3-4, (h/4)-4, 0, vFont, "%01X", n);
        zValue = Rectangle2(valuesLayer, 0, 0, yellowBrush, zx, zy+4, 2*w/3-4, (h/4)-4, 0, vFont, "%01X", z);
        cValue = Rectangle2(valuesLayer, 0, 0, yellowBrush, cx, cy+4, 2*w/3-4, (h/4)-4, 0, vFont, "%01X", c);
        vValue = Rectangle2(valuesLayer, 0, 0, yellowBrush, vx, vy+4, 2*w/3-4, (h/4)-4, 0, vFont, "%01X", v);


        } else {

                // Outer Rectangle
                outer = Rectangle2(0, 0, blackPen, cpsrBrush, x, y, w+4, h);                //Draw outer rectangle

                // Labels
                nLabel = Rectangle(0, 0, 0, 0, nx, y, 0, 0, w/4, h/3, 0, lFont, "N");
                zLabel = Rectangle(0, 0, 0, 0, zx, y, 0, 0, w/4, h/3, 0, lFont, "Z");
                cLabel = Rectangle(0, 0, 0, 0, cx, y, 0, 0, w/4, h/3, 0, lFont, "C");
                vLabel = Rectangle(0, 0, 0, 0, vx, y, 0, 0, w/4, h/3, 0, lFont, "V");

                // Values
        nValue = Rectangle2(valuesLayer, 0, 0, yellowBrush, nx+4, ny, (w/4)-4, 2*h/3-4, 0, vFont, "%01X", n);
        zValue = Rectangle2(valuesLayer, 0, 0, yellowBrush, zx+4, zy, (w/4)-4, 2*h/3-4, 0, vFont, "%01X", z);
        cValue = Rectangle2(valuesLayer, 0, 0, yellowBrush, cx+4, cy, (w/4)-4, 2*h/3-4, 0, vFont, "%01X", c);
        vValue = Rectangle2(valuesLayer, 0, 0, yellowBrush, vx+4, vy, (w/4)-4, 2*h/3-4, 0, vFont, "%01X", v);

        }


        //
// update Values
//
num function updateNValue() {
nValue.setTxt("%01X", n);
                return 0;
}

num function updateZValue() {
zValue.setTxt("%01X", z);
                return 0;
}

num function updateCValue() {
cValue.setTxt("%01X", c);
                return 0;
}

num function updateVValue() {
vValue.setTxt("%01X", v);
                return 0;
}

        num function updateValues() {
                updateNValue();
                updateZValue();
                updateCValue();
                updateVValue();
        }



        //
// mouse enter exit event handlers for each n,z,c and v
//
when nValue ~> eventEE(num enter, num x, num y) {
nValue.setBrush(enter ? whiteBrush : yellowBrush);
                return 0;
}
```

```
when zValue ~> eventEE(num enter, num x, num y) {
zValue.setBrush(enter ? whiteBrush : yellowBrush);
                return 0;
}

when cValue ~> eventEE(num enter, num x, num y) {
cValue.setBrush(enter ? whiteBrush : yellowBrush);
                return 0;
}

when vValue ~> eventEE(num enter, num x, num y) {
vValue.setBrush(enter ? whiteBrush : yellowBrush);
                return 0;
}

//
// mouse left button event handler
//
when nValue ~> eventMB(num down, num flags, num x, num y) {
if (down) {
                        if (flags) {
        n = (n + 1) % 2;
                        }
updateNValue();
}
                return 0;
}

when zValue ~> eventMB(num down, num flags, num x, num y) {
if (down) {
                        if (flags) {
        z = (z + 1) % 2;
                        }
updateZValue();
}
                return 0;
}

when cValue ~> eventMB(num down, num flags, num x, num y) {
if (down) {
                        if (flags) {
        c = (c + 1) % 2;
                        }
updateCValue();
}
                return 0;
}

when vValue ~> eventMB(num down, num flags, num x, num y) {
if (down) {
                        if (flags) {
        v = (v + 1) % 2;
                        }
updateVValue();
}
                return 0;
}

//
// set values
//
function setNValue(num val) {
n = val;
updateNValue();
}

function setZValue(num val) {
z = val;
updateZValue();
}

function setCValue(num val) {
c = val;
updateCValue();
}

function setVValue(num val) {
v = val;
updateVValue();
}

        //
// set next values
```

```
//
function setNNValue(num val) {
nN = val;
}

function setNZValue(num val) {
nZ = val;
}

function setNCValue(num val) {
nC = val;
}

function setNVValue(num val) {
nV = val;
}


          function update() {
                    n = nN;
                    z = nZ;
                    c = nC;
                    v = nV;
                    invalid = 0;
                    updateValues();
          }


          function setInvalid(num val) {
                    invalid = val;
                    if(invalid){
                              nValue.setTxt("X");
                              zValue.setTxt("X");
                              cValue.setTxt("X");
                              vValue.setTxt("X");
                    }
          }

          //
// setOpacity
//
function setOpacity(num opacity) {
outer.setOpacity(opacity);
nLabel.setOpacity(opacity);
zLabel.setOpacity(opacity);
cLabel.setOpacity(opacity);
vLabel.setOpacity(opacity);
nValue.setOpacity(opacity);
zValue.setOpacity(opacity);
cValue.setOpacity(opacity);
vValue.setOpacity(opacity);
}

num hmode = 0;

//
// reset
//
function reset() {
          n = 0;
                    z = 0;
                    c = 0;
                    v = 0;
updateValues();
                    setInvalid(1);
}



}

// eof


//
// default.vin
//
// Simulation of the ARM
// Written by Rónán Dowling-Cullen, Final Year Computer Science 2018/19DLX
// Building on the simulation of the DLX written by Edsko de Vries, Summer 2003

//
// pen widths
//
const num THIN = 1;
const num MEDIUM = 3;
const num THICK = 5;

//
//colours (NB: web-safe)
//
const num DARK_BLUE = rgba(0.0, 0.0, 0.625);
const num LIGHT_BLUE = rgba(0.75, 1.0, 1.0);
const num PURPLE = rgba(0.75, 0.625, 0.75);
const num BORDEAU = rgba(0.375, 0.0, 0.0);
const num MARINE = rgba(0.375, 0.625, 0.375);
const num LIGHT_YELLOW = rgba(1.0, 1.0, 0.75);
const num ORANGE = 0xffa500;

//
// Pens
//
Pen blackPen = SolidPen(0, 1, BLACK);                                     // {joj 2/10/16}
Pen darkGrayPen = SolidPen(0, 0, GRAY64);                      // {joj 25/7/10}
Pen redPen = SolidPen(0, 0, RED);                                              // {joj 2/10/16}
Pen whitePen = SolidPen(0, 0, WHITE);                                     // {joj 2/10/16}

//
// Brushes
//
Brush blackBrush = SolidBrush(BLACK);                                     // {joj 2/10/16}
Brush darkGrayBrush = SolidBrush(GRAY64);
Brush lightBlueBrush = SolidBrush(LIGHT_BLUE);
Brush gray192Brush = SolidBrush(GRAY192);                      // {joj 17/10/16}
Brush gray224Brush = SolidBrush(GRAY224);                      // {joj 17/10/16}
Brush marineBrush = SolidBrush(MARINE);
Brush redBrush = SolidBrush(RED);                                              // {joj 2/10/16}
Brush whiteBrush = SolidBrush(WHITE);                                     // {joj 2/10/16}
Brush yellowBrush = SolidBrush(YELLOW);                                   // {joj 2/10/16}

// eof

//
// execution.vin
//
// Simulation of the ARM
// Written by Rónán Dowling-Cullen, Final Year Computer Science 2018/19DLX
// Building on the simulation of the DLX written by Edsko de Vries, Summer 2003

AnimPipe mux1Src, mux3Src, mux4Src;

num blockingCond = 0;                       //Indicates conditional unexecution
num zero;

num newPC;

//num branchMissPredict;

AnimPipe newMux1Src;


num mulStall = 0;

//
// index for tag or -1 if none
//
num function btbIndex(num pc) {
for (num lp1 = 0; lp1 < 2; lp1++)
if (btbPC[lp1].value == pc)
return lp1;
return -1;
}

//
// ifExec
//
// 80 ticks per clock cycle (40 + 40)
//
function ifExec() {



          if ((stall == NO_STALL) || (stall == CTRL_STALL)) {
fork(regPC.update());
fork(regFile[15].update());
im.setActive(regPC.newValue);
}



        wait(8);

        if (instrIsBranch(regID.vIns) && (regID.vCond != def) && ((regEX.vSBit == "S") || (regEX.vIns >= CMN && regEX.vIns <=
TSTi))) {
        } else {
//
// look for PC is branch target buffer
//
if ((bpMode == BRANCH_PREDICTION) && (btbIndex(regPC.value) != -1)) {
btbLast = btbIndex(regPC.value);
regPC.setNewValue(btbPPC[btbLast].value);
mux1Src = apBTB_MUX1;
} else {
regPC.setNewValue((regPC.value + 4) & 0x7F);
mux1Src = apADD4_MUX1;
}

        }

        //Update PC visualised as R15 in register file
        regFile[15].setNewValue(regPC.newValue);

regPC1.setNewValue(regPC.value);
regID.setNewInstruction(im.instruction[regPC.value / 4]);

        //Convert shift ops to move ops
        if((regID.vIns >= LSL && regID.vIns <= ROR) || (regID.vIns >= LSLi && regID.vIns <= RORi)){
                regID.vIns2 = regID.vIns;
                regID.vIns = MOV;
                regID.vRs3 = regID.vRs2;
                regID.vRs2 = regID.vRs1;
        }

wait(8);

        fork(apPC_PC1.animate(64));
        fork(apPC_IM.animate(24));
        fork(apPC_ADD4.animate(24));

if ((bpMode == BRANCH_PREDICTION) && (instrIsBranch(regID.vIns))) {
                if (stall == CTRL_STALL) {

                        apPC_MUX2.animate(12);

                } else {

        apPC1_MUX2.animate(12);

                }
apMUX2_BTB.animate(12);
} else {
wait(24);
}

//
// Second half of the clock cycle
//
fork(apIM_ID.animate(40));

if ((bpMode == BRANCH_PREDICTION) && (btbIndex(regPC.value) != -1)) {
btbPC[btbIndex(regPC.value)].highlight(bhighlight);
btbPPC[btbIndex(regPC.value)].highlight(bhighlight);
}

txtIM_ID.setTxt(regID.getNewInstrTxt());
txtIM_ID.setOpacity(1, 16, 1, 1);

mux1Src.animate(16);
apMUX1_PC.animate(8);

}

//
// set mux1Src, mux3Src and regPC.newValue and values for BTB if necessary
//
function calcNewPC() {

// AnimPipe newMux1Src;
```

```
if (instrIsBranch(regID.vIns)){
                newPC = regPC1.value + 4;
                if((regID.vCond == def) || ((regEX.vSBit != "S") && (regEX.vIns <= CMN || regEX.vIns >= TSTi))) {
                // Stall to allow for condition branching on instruction one step ahead
        // conditional branch
                        Pen pen = controlHLPen;
                        mux3Src = apADDi_MUX3;
                mux4Src = apMUX3_MUX4;

                        if(testCondition(regID.vCond)){
                                if(regID.vIns == B || regID.vIns == BL) {
                                newPC = (regPC1.value + regID.vRs2) & 0x7F;
                                }
                        } else {
                                newPC = regPC1.value + 4;
                        }
                newMux1Src = apMUX3_MUX1;

                }

        }



}

function updBTB() {

        //branchMissPredict = 0;

        if (newPC != regPC.value) {

//branchMissPredict = 1;

                regPC.setNewValue(newPC);
                regFile[15].setNewValue(newPC);           // Update PC in regfile
mux1Src = newMux1Src;

//
// update branch target buffer
//
if (bpMode == BRANCH_PREDICTION) {

//
// remove "branch" entry if branch doesn't branch
//
if (newPC == regPC1.value + 4) {

if (btbIndex(regPC1.value) >= 0)
btbPC[btbIndex(regPC1.value)].setInvalid(1);

} else {

//
// re-use entry if present
//
if (btbIndex(regPC1.value) >= 0)
btbLast = btbIndex(regPC1.value)
else
btbLast = (btbLast) ? 0 : 1;

btbPC[btbLast].setNewValue(regPC1.value);
btbPC[btbLast].setInvalid(0);
btbPC[btbLast].useTag = 0;
btbPPC[btbLast].setNewValue(newPC);

}

}

}

}

//
// detectStall
//
function detectStall() {

stall = NO_STALL;
updateBTB = 0;

// RAW hazards
```

```
if (afMode == ALU_INTERLOCK) {

if (instrOpTypeRdt(regEX.vIns) == OP_TYPE_REG) {
if ((instrOpTypeRs1(regID.vIns) == OP_TYPE_REG) && (regID.vRs1 == regEX.vRdt))
stall = DATA_STALL;
if ((instrOpTypeRs2(regID.vIns) == OP_TYPE_REG) && (regID.vRs2 == regEX.vRdt))
stall = DATA_STALL;
}

if (instrOpTypeRdt(regMA.vIns) == OP_TYPE_REG) {
if ((instrOpTypeRs1(regID.vIns) == OP_TYPE_REG) && (regID.vRs1 == regMA.vRdt))
stall = DATA_STALL;
if ((instrOpTypeRs2(regID.vIns) == OP_TYPE_REG) && (regID.vRs2 == regMA.vRdt))
stall = DATA_STALL;
}

}

// RAW hazards (stores)
if ((sfMode == STORE_INTERLOCK) && (regID.vIns == STR || regID.vIns == STR)) {

//
// NB: ST stores the value of Rdt into memory (Rs1 and Rs2 are used to specify the memory address)
//
if ((instrOpTypeRdt(regEX.vIns) == OP_TYPE_REG) && (regEX.vRdt == regID.vRdt))
stall = DATA_STALL;
if ((instrOpTypeRdt(regMA.vIns) == OP_TYPE_REG) && (regMA.vRdt == regID.vRdt))
stall = DATA_STALL;

}


// Load hazards
if ((liMode == LOAD_INTERLOCK) && (regEX.vIns == LDR || regEX.vIns == LDRi)) {
if ((instrOpTypeRs1(regID.vIns) == OP_TYPE_REG) && (regID.vRs1 == regEX.vRdt))
stall = DATA_STALL;
if ((instrOpTypeRs2(regID.vIns) == OP_TYPE_REG) && (regID.vRs2 == regEX.vRdt))
stall = DATA_STALL;
}

        //Dealling with Multiplication Stalls

        //Dealing with stalls due to MUL
        if (stall == NO_STALL && mulStall == 0 && (regID.vIns == MUL)) {
                //Based on 8 bit Booth's algorithm calculate the number of stalls needed
                num Rm = regFile[regID.vRs1].value;
                num Rs = regFile[regID.vRs2].value

                if((Rm == 0xFF || Rm == 0x00)
                && (Rs == 0xFF || Rs == 0x00)) {
                        mulStall = 0;
                } else {
                        mulStall = 1;
                }
        } else if (mulStall > 0){
                mulStall--;
                stall = MUL_STALL;
        }


// control hazards (real detection for control hazards is done in calcNewPC)
        if ((stall == NO_STALL) && instrIsBranch(regID.vIns) && (newPC != regPC.value)) {
                if (bpMode == BRANCH_PREDICTION) {
                        updateBTB = 1;
                } else if (bpMode == BRANCH_INTERLOCK) {
        stall = CTRL_STALL;
                }
}

        // Stall to allow for condition branching on instruction one step ahead
        if ((stall == NO_STALL) && instrIsBranch(regID.vIns) && (regID.vCond != def) && ((regEX.vSBit == "S") || (regEX.vIns >=
CMN && regEX.vIns <= TSTi))) {
                stall = CTRL_STALL;
}


if (stall == DATA_STALL) {
animClock.setStall(1, 0);
} else if (stall == CTRL_STALL) {
animClock.setStall(1, 1);
}

}
```

```
function sendBTBOperands() {
mux4Src.animate(18);
apMUX4_BTB.animate(6);
}

//
// idExec
//
function idExec() {


if (stall == NO_STALL) {
        fork(regPC1.update());
fork(regID.update());
}

if (updateBTB && (bpMode == BRANCH_PREDICTION))        {
fork(btbPC[btbLast].update());

fork(btbPPC[btbLast].update());

}



wait(16);

fork(apID_EX.animate(64));


        if(instrIsBranch(regID.vIns)) {

                if (regID.vIns == BL) {

                        regB.setNewValue(regPC1.value);
                }

                if((regID.vCond == def) || ((regEX.vSBit != "S") && (regEX.vIns <= CMN || regEX.vIns >= TSTi))) {

                        fork(apPC1_ADDi.animate(16));
        fork(apID_ADDi.animate(16));
        wait(12);
        txtID_ADDi.setTxt("%02X", regID.vRs1);
        txtID_ADDi.setOpacity(1);
        wait(4);
                        fork(apADDi_MUX3.animate(8));
                        txtADDi_MUX3.setTxt("%02X", (regPC1.value + regID.vRs1) & 0xff);
        txtADDi_MUX3.setOpacity(1, 8, 1, 0);
                        //Animate blocking/allowing write if condition
                        if(regID.vCond != def) {
                                if(testCondition(regID.vCond)) {
                                        conditionMuxB.setPen(controlMuxHLPen);
                                } else {
                                        conditionMuxB.setPen(muxPen);
                                }
                                cpsrEXB.setPen(controlHLPen);

                                if(!cpsr0.invalid){
                                        cpsrOUT0B.setPen(controlHLPen);
                                } else if (!cpsr1.invalid){
                                        cpsrOUT1B.setPen(controlHLPen);
                                } else {
                                        cpsrCPSRB.setPen(controlHLPen);
                                }
                        }

                } else {


                        wait(24);


                }


} else {

wait(24);

};

wait(9); // register file now updated
```

```
//
// second half of the clock cycle
//


    if (instrIsBranch(regID.vIns)) {
calcNewPC();

        if(((regID.vCond == def) || ((regEX.vSBit != "S") && (regEX.vIns <= CMN || regEX.vIns >= TSTi))) &&
testCondition(regID.vCond)) {
            txtMUX3_MUX4.setTxt("%02X", newPC);

            txtMUX3_MUX4.setOpacity(1, 8, 1, 0);
        fork(sendBTBOperands());
        }
    }

detectStall();
    if (instrIsBranch(regID.vIns) && (stall != DATA_STALL))
        updBTB();


if (stall == NO_STALL || stall == MUL_STALL) {
regEX.setNewValue(regID.vIns, regID.vSBit, regID.vCond, regID.vRdt, regID.vRs1, regID.vRs2, regID.vIns2, regID.vRs3);
} else {
regEX.setNewValue(STALL, " ", 0, 0, 0, 0, STALL, 0);
}
wait(7);


// Send source registers to A and B
if ((instrOpTypeRdt(regID.vIns) == OP_TYPE_REG || instrHasNoDstRR(regID.vIns) || instrHasNoDstRI(regID.vIns)) && (regID.vIns != B) &&
(regEX.vIns != MUL || stall != MUL_STALL)){//!instrIsBranch(regID.vIns)) { //Rdt is type reg or it's a compare instruction

if(instrOpTypeRs1(regID.vRs1) != OP_TYPE_IMM)
            regFile[regID.vRs1].highlight(bhighlight);
if(instrOpTypeRs1(regID.vIns) != OP_TYPE_UNUSED)       //Don't set if not in use
            regA.setNewValue(regFile[regID.vRs1].value);

// decision facilitated by the opcode format and does not need a MUX
if (instrOpTypeRs2(regID.vIns) == OP_TYPE_REG) {
regFile[regID.vRs2].highlight(bhighlight);
regB.setNewValue(regFile[regID.vRs2].value);
} else if (regID.vIns != BL) {
if(instrOpTypeRdt(regID.vRdt) != OP_TYPE_UNUSED)
                regFile[regID.vRdt].highlight(bhighlight);
regB.setNewValue(regFile[regID.vRdt].value);
}

        if(instrOpTypeRs1(regID.vIns) != OP_TYPE_UNUSED) {     //Don't animate if not in use
    txtRF_A.setTxt("R%d:%02X", regID.vRs1, regFile[regID.vRs1].value);
    txtRF_A.setOpacity(1);

            fork(apRF_A.animate(24));
        }


//
// read Register file B port
// don't read if immediate addressing
// don't read if LDR
// read destination register if STR
// otherwise read source 2 register
//
if ((!instrIsArRI(regID.vIns)) && (regID.vIns != LDR && regID.vIns != LDRi) && (!instrHasNoOp1RI(regID.vIns))) {
num vr = (regID.vIns == STR || regID.vIns == STRi) ? regID.vRdt : regID.vRs2;
if(regID.vIns == BL) {
                apPC1_MUX5.animate(18);
    apMUX5_B.animate(6);
            } else if(instrOpTypeRs2(regID.vIns) != OP_TYPE_UNUSED) {         //Don't animate if not in use
                txtRF_MUX5.setTxt("R%d:%02X", vr, regFile[vr].value);
txtRF_MUX5.setOpacity(1);
    apRF_MUX5.animate(18);
    apMUX5_B.animate(6);
            }
        }





}
```

```
    //Flush regID if Branch Instr and not conditionally unexecuted
    if(instrIsBranch(regID.vIns)){
        if(bpMode == BRANCH_PREDICTION){
            if((btbIndex(regPC1.value) != -1) && !testCondition(regID.vCond)){
                regID.setNewValue(STALL, " ", 0, 0, 0, 0, STALL, 0);
            }
            if((btbIndex(regPC1.value) == -1) && testCondition(regID.vCond)){
                regID.setNewValue(STALL, " ", 0, 0, 0, 0, STALL, 0);
            }
        }
    }


}

//
// exExec
//
function exExec() {
            // {joj 14/11/17}

AnimPipe mux6Src, mux7Src, mux8Src;
num op1, op2, op3;



    if (stall != MUL_STALL) {

    fork(regEX.update());
}

    if (!instrIsNop(regEX.nIns)) {
fork(regA.update());
fork(regB.update());
}



wait(8);

regMA.setNewValue(regEX.vIns, regEX.vSBit, regEX.vCond, regEX.vRdt, regEX.vRs1, regEX.vRs2, regEX.vIns2, regEX.vRs3);

if (instrOpTypeRdt(regEX.vIns) == OP_TYPE_REG || instrHasNoDstRR(regEX.vIns) || instrHasNoDstRI(regEX.vIns)) { //Rdt is type reg or
it's a compare instruction

// select correct source for operand 1
        if (afMode == ALU_FORWARDING) {

if (regOUT0.tagMatches(regEX.vRs1)) {
mux6Src = apOUT0_MUX6;
op1 = regOUT0.value;
} else if (regOUT1.tagMatches(regEX.vRs1)) {
mux6Src = apOUT1_MUX6;
op1 = regOUT1.value;
} else {
mux6Src = apA_MUX6;
op1 = regA.value;
}

} else if(regEX.vIns != B && regEX.vIns != BL) {

mux6Src = apA_MUX6;
op1 = regA.value;

}

// Choose correct source for operand 2
        if(regEX.vIns == BL) {
            mux7Src = apB_MUX7;
op2 = regB.value;

        } else if (instrOpTypeRs2(regEX.vIns) == OP_TYPE_IMM && !instrIsBranch(regEX.vIns)) {

mux7Src = apEX_MUX7;
op2 = regEX.vRs2;

} else if (afMode == ALU_FORWARDING && regEX.vIns != BL) {        //If it's BL op2 <- regB (the address of the BL instruction)

if (regOUT0.tagMatches(regEX.vRs2)) {
mux7Src = apOUT0_MUX7;
op2 = regOUT0.value;
} else if (regOUT1.tagMatches(regEX.vRs2)) {
mux7Src = apOUT1_MUX7;
op2 = regOUT1.value;
} else {
mux7Src = apB_MUX7;
op2 = regB.value;
}

} else {

mux7Src = apB_MUX7;
op2 = regB.value;

}

        // Choose correct source for operand 3
        if (instrOpTypeRs3(regEX.vIns,regEX.vIns2) == OP_TYPE_IMM) {

op3 = regEX.vRs3;

} else if (afMode == ALU_FORWARDING && regEX.vIns != BL) {        //If it's BL op2 <- regB (the address of the BL instruction)

if (regOUT0.tagMatches(regEX.vRs3)) {
op3 = regOUT0.value;
} else if (regOUT1.tagMatches(regEX.vRs3)) {
op3 = regOUT1.value;
} else {
op3 = regFile[regEX.vRs3].value;
}

} else {

mux7Src = apB_MUX7;
op3 = regFile[regEX.vRs3].value;

}

num result = instrExecute(regEX.vIns, regEX.vSBit, regEX.vCond, op1, op2, regEX.vIns2, op3, cpsr.n, cpsr.z, cpsr.c, cpsr.v);

if (instrIsLoadOrStore(regEX.vIns)) {
regOUT0.setNewTag(-1);                                            // OUT0 contains the EA, tag should be -1
} else if (regEX.vIns == BL) {
            regOUT0.setNewTag(14);                       //If its a branch and link instruction tag it as LR
        } else {
regOUT0.setNewTag(regEX.vRdt);
        }

        //If condition fails result is old value of Rd
        if (testCondition(regEX.vCond)) {
            regOUT0.setNewValue(result);
        } else {
            regOUT0.setNewTag(-2);
        }

        //Dealing with early output on MUL Stall
        if(mulStall > 0){
            regOUT0.setNewTag(-1);
        }




}

// For stores, send correct operand to SMDR
if ((regEX.vIns == STR || regEX.vIns == STRi)) { // {joj}
if (sfMode == FORWARDING_TO_SMDR) {
if (regOUT0.tagMatches(regEX.vRdt)) {
mux8Src = apOUT0_MUX8;
regSMDR.setNewValue(regOUT0.value);
} else if (regOUT1.tagMatches(regEX.vRdt)) {
mux8Src = apOUT1_MUX8;
regSMDR.setNewValue(regOUT1.value);
} else {
mux8Src = apB_MUX8;
regSMDR.setNewValue(regFile[regEX.vRdt].value);
}
} else {
mux8Src = apB_MUX8;
regSMDR.setNewValue(regFile[regEX.vRdt].value);
}
}
```

```
wait(8);

fork(apEX_MA.animate(64));

if ((regEX.vIns == STR || regEX.vIns == STRi)) {
fork(mux8Src.animate(24));
        //Deal with barrel shift ops
        if(regEX.vIns2 != NOP){
            if(instrIsShI(regEX.vIns2)){
                txtEX_BS.setTxt("%02X", op3);
            } else {
                txtRF_BS.setTxt("R%d:%02X", regEX.vRs3, regFile[regEX.vRs3].value);
                txtRF_BS.setOpacity(1);
            }
        }
    }

if ((instrOpTypeRdt(regEX.vIns) == OP_TYPE_REG || instrHasNoDstRR(regEX.vIns) || instrHasNoDstRI(regEX.vIns)) && regEX.vIns != STALL
&& regEX.vIns != B) { //Rdt is type reg or it's a compare instruction
if (mux6Src != 0 && (instrOpTypeRs1(regEX.vIns) != OP_TYPE_UNUSED))
fork(mux6Src.animate(24));
if (mux7Src == apEX_MUX7) {
txtEX_MUX7.setTxt("%02X", op2);
txtEX_MUX7.setOpacity(1);
}
        if(instrOpTypeRs2(regEX.vIns) != OP_TYPE_UNUSED)
            fork(mux7Src.animate(24));

        //Deal with barrel shift ops
        if(regEX.vIns2 != NOP){
            if(instrIsShI(regEX.vIns2)){
                fork(apEX_BS.animate(24));
                txtEX_BS.setTxt("%02X", op3);
            } else {
                fork(apRF_BS.animate(24));
                txtRF_BS.setTxt("R%d:%02X", regEX.vRs3, regFile[regEX.vRs3].value);
                txtRF_BS.setOpacity(1);
            }
        }

}

wait(12);
    //Delay visualising Immediate value from EX to wait for animPipe
    if(regEX.vIns2 != NOP){
        if(instrIsShI(regEX.vIns2)){
            txtEX_BS.setOpacity(1);
        }
    }
    wait(12);

//
// Second half of the clock cycle
//

if ((regEX.vIns == STR || regEX.vIns == STRi))
fork(apMUX8_SMDR.animate(40));

if (((instrOpTypeRdt(regEX.vIns) == OP_TYPE_REG || instrHasNoDstRR(regEX.vIns) || instrHasNoDstRI(regEX.vIns))) && regEX.vIns !=
STALL && regEX.vIns != B) { //Rdt is type reg or it's a compare instruction

alu.setTxtOp(regEX.vIns);

if (mux6Src != 0 && (instrOpTypeRs1(regEX.vIns) != OP_TYPE_UNUSED))
fork(apMUX6_OUT0.animate(40));

        if(instrOpTypeRs2(regEX.vIns) != OP_TYPE_UNUSED)
    fork(apMUX7_OUT0.animate(40));


        wait(20);
        if(regEX.vIns != MUL || stall != MUL_STALL) {

            alu.txtResult.setTxt("%02X", result);
     alu.txtResult.setOpacity(1, 20, 1, 0);

        }

        //Animate blocking/allowing write if condition
        if(regEX.vCond != def && testCondition(regEX.vCond) && !instrIsBranch(regEX.vIns)) {
            conditionMux.setPen(controlMuxHLPen);
            cpsrEX.setPen(controlHLPen);
```

```
                                if(!cpsr0.invalid){
                                        cpsrOUT0.setPen(controlHLPen);
                                } else if (!cpsr1.invalid){
                                        cpsrOUT1.setPen(controlHLPen);
                                } else {
                                        cpsrCPSR.setPen(controlHLPen);
                                }
                        }

}



                //Set Validity of regOUT0
if ((instrOpTypeRdt(regEX.vIns) == OP_TYPE_REG || instrHasNoDstRR(regEX.vIns) || instrHasNoDstRI(regEX.vIns)) && (regEX.vIns != B) &&
!(regEX.vIns >= CMN && regEX.vIns <= TSTi)) {
                        regOUT0.setInvalid(0);

} else if (regEX.vIns == NOP || !testCondition(regEX.vCond) || regEX.vIns == B || regEX.vIns == STALL || (regEX.vIns >= CMN &&
regEX.vIns <= TSTi)) {
                        regOUT0.setInvalid(1);
                        regOUT0.setNewTag(-1);
                        regOUT0.setNewValue(0);
                }



                if ((regEX.vIns == NOP || !testCondition(regEX.vCond)) && !instrIsBranch(regEX.vIns)) { //Delay updating label for better
visualisation

                        if(regEX.vCond != def) {
                                blockingCond = 1;
                        }
                        //Animate blocking write if condition
                        if(blockingCond) {
                                conditionMux.setPen(muxPen);
                                cpsrEX.setPen(controlHLPen);
                                if(!cpsr0.invalid){
                                        cpsrOUT0.setPen(controlHLPen);
                                } else if (!cpsr1.invalid){
                                        cpsrOUT1.setPen(controlHLPen);
                                } else {
                                        cpsrCPSR.setPen(controlHLPen);
                                }
                                blockingCond = 0;
                        }
                        wait(40);
                        regOUT0.updateLabel();
                }


}

//
// maExec - Memory Access phase
//
function maExec() {



                if(stall == MUL_STALL) {        //Stall ahead of MUL op
                        regMA.setNewValue(STALL, " ", 0, 0, 0, 0, STALL, 0);
                }

                //Check for coditional nonexecution
                if(regOUT0.tag == -2) {
                        regOUT1.setNewTag(-2);
                }


                fork(regMA.update());



if (instrOpTypeRdt(regMA.nIns) == OP_TYPE_REG || instrHasNoDstRR(regMA.vIns) || instrHasNoDstRI(regMA.vIns) || regEX.vIns == BL) {
//Rdt is type reg or it's a compare instruction
fork(regOUT0.update());
                }

                wait(1);

                //Update CPSR0 if S set or comparison instruction
                if((regMA.vSBit == "S") || (regMA.vIns >= CMN && regMA.vIns <= TST) || (regMA.vIns >= CMNi && regMA.vIns <= TSTi)){
```

```
                        cpsr0.update();
                } else {
                        cpsr0.setInvalid(1);
                }



if (regMA.nIns == STR || regMA.nIns == STRi)
fork(regSMDR.update());

wait(7);

regWB.setNewValue(regMA.vIns, regMA.vSBit, regMA.vCond, regMA.vRdt, regMA.vRs1, regMA.vRs2, regMA.vIns2, regMA.vRs3);
if ((instrOpTypeRdt(regMA.vIns) == OP_TYPE_REG || instrHasNoDstRR(regMA.vIns) || instrHasNoDstRI(regMA.vIns) || regEX.vIns == BL) &&
(regMA.vIns != STR)&& (regMA.vIns != STRi)) { // {joj}

if ((regMA.vIns == LDR) || (regMA.vIns == LDRi)) {
regOUT1.setNewValue(memory[(regOUT0.value / 4) % 16].value);
                                if(regOUT1.tag != -2) {
                regOUT1.setNewTag(regMA.vRdt);
                                }
} else {

regOUT1.setNewValue(regOUT0.value);
regOUT1.setNewTag(regOUT0.tag);
}

regOUT1.setInvalid(0);

}

wait(8);

fork(apMA_WB.animate(64));
if ((regMA.vIns == STR || regMA.vIns == STRi) && (regOUT0.tag != -2)) {


memory[(regOUT0.value / 4) % 16].setNewValue(regSMDR.value);
fork(apSMDR_Mem.animate(24));
apOUT0_Mem.animate(24);
memory[(regOUT0.value / 4) % 16].doubleUpdate(stack[((regOUT0.value / 4) % 16) -8]);

} else if (instrOpTypeRdt(regMA.vIns) == OP_TYPE_REG || instrHasNoDstRR(regMA.vIns) || instrHasNoDstRI(regMA.vIns) || regEX.vIns ==
BL) { //Rdt is type reg or it's a compare instruction

if ((regMA.vIns == LDR || regMA.vIns == LDRi) && (regOUT0.tag != -2)) {
apOUT0_Mem.animate(24);
memory[(regOUT0.value / 4) % 16].highlight(bhighlight);
apMem_MUX9.animate(24);
} else {
apOUT0_MUX9.animate(48);
}
apMUX9_OUT1.animate(16);

}

                if (regMA.vIns == NOP || !testCondition(regMA.vCond)) {          //Delay updating label for better visualisation
                        wait(64);
                        regOUT1.setInvalid(1);
                        regOUT1.updateLabel();
                }

                //Pass CPSR0 to CPSR1 if S set or comparison instruction
                if((regMA.vSBit == "S") || (regMA.vIns >= CMN && regMA.vIns <= TST) || (regMA.vIns >= CMNi && regMA.vIns <= TSTi)){
                        cpsr1.setNNValue(cpsr0.n);
                        cpsr1.setNZValue(cpsr0.z);
                        cpsr1.setNCValue(cpsr0.c);
                        cpsr1.setNVValue(cpsr0.v);
                }

}

//
// wbExec - write back phase
//
function wbExec() {


fork(regWB.update());


wait(1); //To guarantee regWB updated in time for following if statement
```

```
if ((instrOpTypeRdt(regWB.vIns) == OP_TYPE_REG || instrHasNoDstRR(regWB.vIns) || instrHasNoDstRI(regWB.vIns) || regWB.vIns == BL)){
fork(regOUT1.update());
    }

    //Update CPSR1 if S set or comparison instruction
    if((regWB.vSBit == "S") || (regWB.vIns >= CMN && regWB.vIns <= TST) || (regWB.vIns >= CMNi && regWB.vIns <= TSTi)){
        cpsr1.update();
    } else {
        cpsr1.setInvalid(1);
    }

    wait(7);
if ((regWB.vIns != STRi) && (regWB.vIns != STR) && (regWB.vIns != B) && (instrOpTypeRdt(regWB.vIns) == OP_TYPE_REG ||
instrHasNoDstRR(regWB.vIns) || instrHasNoDstRI(regWB.vIns) || regWB.vIns == BL) && (regOUT1.tag != -2)) {

        //Update CPSR1 if S set or comparison instruction
        if((regWB.vSBit == "S") || (regWB.vIns >= CMN && regWB.vIns <= TST) || (regWB.vIns >= CMNi && regWB.vIns <=
TSTi)){
            cpsr.setNNValue(cpsr1.n);
            cpsr.setNZValue(cpsr1.z);
            cpsr.setNCValue(cpsr1.c);
            cpsr.setNVValue(cpsr1.v);
        }

        regFile[regOUT1.tag].setNewValue(regOUT1.value);

wait(8);

fork(apOUT1_RF.animate(24));
        if((regWB.vSBit == "S") || (regWB.vIns >= CMN && regWB.vIns <= TST) || (regWB.vIns >= CMNi && regWB.vIns <=
TSTi)){
            fork(apCPSR1_RF.animate(24));
        }

//
// Second half of the clock cycle
//
regFile[regOUT1.tag].update();
        //Update CPSR1 if S set or comparison instruction
        if((regWB.vSBit == "S") || (regWB.vIns >= CMN && regWB.vIns <= TST) || (regWB.vIns >= CMNi && regWB.vIns <=
TSTi)){
            cpsr.update();
        }

        //Update the stack pointer visualisation
        if(regOUT1.tag == 13){
            sp.moveSP(regFile[13].value);
        }
wait(19);

} else {

wait(67);

}

if (regWB.vIns != STALL && regWB.vIns != EMPTY) {
instrCount++;
statIC.setTxt("%4d", instrCount);
}
tickCount++;
statTE.setTxt("%4d", tickCount);

}

//
// nonPipelinedBranch
//
function nonPipelinedBranch() {
fork(apID_ADDi.animate(24));
    apPC_PC1.animate(12);

fork(apPC1_ADDi.animate(12));
    wait(12);



    if (instrIsBranch(regID.vIns)) {
if ((regFile[regID.vRs1].value == 0) == (regID.vIns == B)) {
apADDi_MUX3.animate(20);
regPC.setNewValue((regPC.value + regID.vRs2) & 0x7F);
} else {
regPC.setNewValue((regPC.value + 4) & 0x7F);
```

```
}
apMUX3_MUX1.animate(14);
} else {
regPC.setNewValue((regPC.value + 4) & 0x7F);
apMUX3_MUX1.animate(14);
}

    regFile[15].setNewValue(regPC.newValue);        // Update PC in regfile

apMUX1_PC.animate(6);

}

//
// execNonPipelined()
//
function execNonPipelined() {


num regA, regB, result, smdr;

// Update registers

// IF

regPC.update();
    regFile[15].update();        // Update PC in regfile
im.setActive(regPC.newValue);

apPC_IM.animate(24);

apIM_ID.animate(40);
regID.setNewInstruction(im.instruction[regPC.value / 4]);
txtIM_ID.setTxt(regID.getNewInstrTxt());
txtIM_ID.translate(60/2+70, 0, 20, 1, 0);

    /// ID

regID.update();


if ((instrOpTypeRs2(regID.vIns) == OP_TYPE_IMM) && (instrOpTypeRdt(regID.vIns) == OP_TYPE_REG))
fork(apID_EX.animate(64));

fork(nonPipelinedBranch());

wait(24);

// Second half of the clock cycle

//} else
    if (instrOpTypeRdt(regID.vIns) == OP_TYPE_REG) {

regA = regFile[regID.vRs1].value;
        if(instrOpTypeRs1(regID.vRs1 != OP_TYPE_IMM))
    regFile[regID.vRs1].highlight(bhighlight);

txtRF_A.setTxt("R%d:%02X", regID.vRs1, regFile[regID.vRs1].value);
txtRF_A.setOpacity(1);
fork(apRF_A.animate(40));

if ((instrOpTypeRs2(regID.vIns) == OP_TYPE_REG) || (regID.vIns == STR || regID.vIns == STRi)) {

if (instrOpTypeRs2(regID.vIns) == OP_TYPE_IMM) {
regB = regFile[regID.vRdt].value;
                if(instrOpTypeRdt(regID.vIns != OP_TYPE_UNUSED))
    regFile[regID.vRdt].highlight(bhighlight);
} else {
regB = regFile[regID.vRs2].value;
regFile[regID.vRs2].highlight(bhighlight);
}

//
// read Register file B port
// don't read if immediate addressing
// don't read if LDR
// read destination register if STR
// otherwise read source 2 register
//

if ((!instrIsArRI(regID.vIns)) && (regID.vIns != LDR && regID.vIns != LDRi)) {
num vr = (regID.vIns == STR || regID.vIns == STRi) ? regID.vRdt : regID.vRs2;
txtRF_MUX5.setTxt("R%d:%02X", vr, regFile[vr].value);
```

```
txtRF_MUX5.setOpacity(1);
apRF_MUX5.animate(20);
apMUX5_B.animate(20);
}

} else {

wait(40);

}

} else {

wait(40);

}

/// EX


if (instrOpTypeRdt(regID.vIns) == OP_TYPE_REG)
alu.setTxtOp(regID.vIns);
if (regID.vIns == STR || regID.vIns == STRi) {

                      //Deal with barrel shift ops
                      if(regEX.vIns2 != NOP){
                                 if(instrIsShI(regID.vIns2)){
                                            fork(apEX_BS.animate(24));
                                            txtEX_BS.setTxt("%02X", regID.vRs3);
                                            txtEX_BS.setOpacity(1);
                                 } else {
                                            fork(apRF_BS.animate(24));
                                            txtRF_BS.setTxt("R%d:%02X", regID.vRs3, regFile[regID.vRs3].value);
                                            txtRF_BS.setOpacity(1);
                                 }
                      }

fork(apB_MUX8.animate(40));
fork(apA_MUX6.animate(40));
txtEX_MUX7.setTxt("%02X", regID.vRs2);
txtEX_MUX7.setOpacity(1);
apEX_MUX7.animate(40);

fork(apMUX8_SMDR.animate(40));
fork(apMUX7_OUT0.animate(40));
apMUX6_OUT0.animate(40);

smdr = regFile[regID.vRdt].value;
result = instrExecute(regID.vIns, regID.vSBit, regID.vCond, regA, regID.vRs2, regID.vIns2, regID.vRs3, cpsr.n, cpsr.z, cpsr.c,
cpsr.v);



} else if (instrOpTypeRdt(regID.vIns) == OP_TYPE_REG) {

fork(apA_MUX6.animate(40));

                      //Deal with barrel shift ops
                      if(regEX.vIns2 != NOP){
                                 if(instrIsShI(regID.vIns2)){
                                            fork(apEX_BS.animate(24));
                                            txtEX_BS.setTxt("%02X", regID.vRs3);
                                            txtEX_BS.setOpacity(1);
                                 } else {
                                            fork(apRF_BS.animate(24));
                                            txtRF_BS.setTxt("R%d:%02X", regID.vRs3, regFile[regID.vRs3].value);
                                            txtRF_BS.setOpacity(1);
                                 }
                      }

if (instrOpTypeRs2(regID.vIns) == OP_TYPE_IMM) {
txtEX_MUX7.setTxt("%02X", regID.vRs2);
txtEX_MUX7.setOpacity(1);
apEX_MUX7.animate(40);
result = instrExecute(regID.vIns, regID.vSBit, regID.vCond, regA, regID.vRs2, regID.vIns2, regID.vRs3, cpsr.n, cpsr.z, cpsr.c,
cpsr.v);
} else {
apB_MUX7.animate(40);
result = instrExecute(regID.vIns, regID.vSBit, regID.vCond, regA, regB, regID.vIns2, regFile[regID.vRs3].value, cpsr.n, cpsr.z,
cpsr.c, cpsr.v);
}

fork(apMUX7_OUT0.animate(40));
```

```
apMUX6_OUT0.animate(40);

} else {

wait(80);

}

/// MA



           if(testCondition(regID.vCond)){

                      if(regID.vCond != def){
                                 conditionMux.setPen(controlMuxHLPen);
                                 cpsrEX.setPen(controlHLPen);
                                 cpsrCPSR.setPen(controlHLPen);
                      }


           if (regID.vIns == LDR || regID.vIns == LDRi) {
            apOUT0_Mem.animate(20);
            memory[((result) / 4) % 16].highlight(bhighlight);
            apMem_MUX9.animate(20);
           apMUX9_OUT1.animate(40);
           result = memory[((result) / 4) % 16].value;
           } else if (regID.vIns == STR || regID.vIns == STRi) {
           fork(apSMDR_Mem.animate(20));
           apOUT0_Mem.animate(20);
           memory[((result) / 4) % 16].setNewValue(smdr);
           memory[((result) / 4) % 16].doubleUpdate(stack[(((result) / 4) % 16) - 8]);
           } else if (instrOpTypeRdt(regID.vIns) == OP_TYPE_REG) {
            apOUT0_MUX9.animate(40);
            apMUX9_OUT1.animate(40);
           } else {
            wait(80);
           }

           /// WB

           regFile[0].unHighlight();
           regFile[1].unHighlight();
           regFile[2].unHighlight();
           regFile[3].unHighlight();

           if ((instrOpTypeRdt(regID.vIns) == OP_TYPE_REG) && (regID.vIns != STR) && (regID.vIns != STRi)) {
            apOUT1_RF.animate(40);
            regFile[regID.vRdt].setNewValue(result);
            regFile[regID.vRdt].update();
            wait(19);
           } else {
            wait(75);
           }

           } else {

                      conditionMux.setPen(muxPen);
                      cpsrEX.setPen(controlHLPen);
                      cpsrCPSR.setPen(controlHLPen);

           }

tickCount += 5;
instrCount++;
statIC.setTxt("%4d", instrCount);
statTE.setTxt("%4d", tickCount);

}

function exec() {

//
// unhighlight registers
//
regFile[0].unHighlight();
regFile[1].unHighlight();
regFile[2].unHighlight();
regFile[3].unHighlight();

memory[0].unHighlight();
memory[1].unHighlight();
memory[2].unHighlight();
```

```
memory[3].unHighlight();

btbPC[0].unHighlight();
btbPC[1].unHighlight();
btbPPC[0].unHighlight();
btbPPC[1].unHighlight();

//
// Run the individual pipeline phases
//
if (peMode == PIPELINING_ENABLED) {
fork(ifExec());
                      fork(idExec());
fork(exExec());
fork(maExec());
fork(wbExec());
} else {
fork(execNonPipelined());
}

wait(8);
resetWires();

wait((peMode == PIPELINING_ENABLED) ? 72 : 392);

checkPoint();

}

// eof



//
// instructionmemory.vin
//
// Simulation of the ARM
// Written by Rónán Dowling-Cullen, Final Year Computer Science 2018/19DLX
// Building on the simulation of the DLX written by Edsko de Vries, Summer 2003

Brush instrMemBrush = SolidBrush(BORDEAU);
Brush instrMemValueBrush = SolidBrush(WHITE);
Pen lightPen = SolidPen(DOT, 1, rgba(0.75, 0.75, 0.75));

Pen redArrow = SolidPen(SOLID, 1, RED, ARROW60_END);

class Instruction(num _x, num _y, num _w, num _h, num _addr) {

           num x = _x;
           num y = _y;
           num w = _w;
           num h = _h;
           num addr = _addr;

           string vSBit = " ";
num vIns = 0, vIns2 = 0, vCond = 0, vRdt = 0, vRs1 = 0, vRs2 = 0, vRs3 = 0;
num opTypeRdt = 0, opTypeRs1 = 0, opTypeRs2 = 0, opTypeRs3 = 0;
           num shiftOk = 0;
           num sOk = 0;
num clk;

num fw = w / 11;

// Brushes for the individual boxes so we can highlight them one by one
Pen insPen = SolidPen(0, 0, BLACK);
Pen sBitPen = SolidPen(0, 0, BLACK);
Pen condPen = SolidPen(0, 0, BLACK);
Pen rdtPen = SolidPen(0, 0, BLACK);
Pen rs1Pen = SolidPen(0, 0, BLACK);
Pen rs2Pen = SolidPen(0, 0, BLACK);
Pen ins2Pen = SolidPen(0, 0, BLACK);
Pen rs3Pen = SolidPen(0, 0, BLACK);

Brush brush = SolidBrush(WHITE);
Rectangle adr = Rectangle2(valuesLayer, 0, 0, brush, x, y, fw, h, 0, font, "%02X", addr);
Rectangle ins = Rectangle2(valuesLayer, HLEFT, 0, brush, x + fw, y, 2 * fw, h, insPen, font, " NOP");
Rectangle sBit = Rectangle2(valuesLayer, HLEFT, 0, brush, x + 3 * fw, y, 0.5 * fw, h, sBitPen, font, "S");
Rectangle cond = Rectangle2(valuesLayer, HLEFT, 0, brush, x + 3.5 * fw, y, fw, h, condPen, font, "AL");
Rectangle rdt = Rectangle2(valuesLayer, 0, 0, brush, x + 5 * fw, y, fw, h, rdtPen, font, "-");
Rectangle rs1 = Rectangle2(valuesLayer, 0, 0, brush, x + 6 * fw, y, fw, h, rs1Pen, font, "-");
Rectangle rs2 = Rectangle2(valuesLayer, 0, 0, brush, x + 7.2 * fw, y, fw, h, rs2Pen, font, "-");
Rectangle ins2 = Rectangle2(valuesLayer, HLEFT, 0, brush, x + 8.2 * fw, y, 2 * fw, h, ins2Pen, font, " NOP");
Rectangle rs3 = Rectangle2(valuesLayer, 0, 0, brush, x + 10 * fw, y, fw, h, rs2Pen, font, "-");
```

```
Rectangle dot = Rectangle2(valuesLayer, 0, 0, redBrush, x+fw*0.8, y+2, h/2, h/2);
dot.setOpacity(0);

Line arrowDown = Line(valuesLayer, 0, redArrow,
0, 0,
                      x + w + 2, y + h * 0.5,
5, 0,
0, 0,
0, 0
);

Line arrowUp = Line(valuesLayer, 0, redArrow,
0, 0,
                      x - 2, y + h * 0.5,
                      -5, 0,
0, 0,
                      0, 0
);

arrowDown.setOpacity(0);
arrowUp.setOpacity(0);

when adr ~> eventEE(num enter, num, num) {
brush.setSolid(enter ? MARINE : WHITE);
                      return 0;
}

when ins ~> eventEE(num enter, num, num) {
brush.setSolid(enter ? MARINE : WHITE);
insPen.setRGBA(enter ? RED : BLACK);
                      return 0;
}

           when sBit ~> eventEE(num enter, num, num) {
brush.setSolid(enter ? MARINE : WHITE);
sBitPen.setRGBA(enter ? RED : BLACK);
                      return 0;
}

           when cond ~> eventEE(num enter, num, num) {
brush.setSolid(enter ? MARINE : WHITE);
condPen.setRGBA(enter ? RED : BLACK);
                      return 0;
}

when ins2 ~> eventEE(num enter, num, num) {
brush.setSolid(enter ? MARINE : WHITE);
ins2Pen.setRGBA(enter ? RED : BLACK);
                      return 0;
}

when rdt ~> eventEE(num enter, num, num) {
brush.setSolid(enter ? MARINE : WHITE);
if (opTypeRdt != OP_TYPE_UNUSED) {
rdtPen.setRGBA(enter ? RED : BLACK);
} else {
rdtPen.setRGBA(BLACK);
                      }
                      return 0;
}

when rs1 ~> eventEE(num enter, num, num) {
brush.setSolid(enter ? MARINE : WHITE);
if (opTypeRs1 != OP_TYPE_UNUSED) {
rs1Pen.setRGBA(enter ? RED : BLACK);
} else {
rs1Pen.setRGBA(BLACK);
                      }
                      return 0;
}

when rs2 ~> eventEE(num enter, num, num) {
brush.setSolid(enter ? MARINE : WHITE);
if (opTypeRs2 != OP_TYPE_UNUSED) {
rs2Pen.setRGBA(enter ? RED : BLACK);
} else {
rs2Pen.setRGBA(BLACK);
                      }
                      return 0;
}

when rs3 ~> eventEE(num enter, num, num) {
```

```
brush.setSolid(enter ? MARINE : WHITE);
if (opTypeRs3 != OP_TYPE_UNUSED) {
rs3Pen.setRGBA(enter ? RED : BLACK);
} else {
rs3Pen.setRGBA(BLACK);
                      }
                      return 0;
}

num function getOpcode() {
return vIns << 24 | vRdt << 16 | vRs1 << 8 | vRs2;
}

function initRegs(num remember) {

num offset;

ins.setTxt("%c%s", 32, instrNames[vIns]);
cond.setTxt("%c%s", 32, condNames[vCond]);

opTypeRdt = instrOpTypeRdt(vIns);
opTypeRs1 = instrOpTypeRs1(vIns);
opTypeRs2 = instrOpTypeRs2(vIns);
opTypeRs3 = instrOpTypeRs3(vIns,vIns2);

                      shiftOk = shiftAllowed(vIns);
                      sOk = sAllowed(vIns);

                      if(sOk) {
           sBit.setTxt("%c%s", 32, vSBit);
                      } else {
                                 sBit.setTxt(" ");
                      }


                      if(shiftOk) {
            ins2.setTxt("%c%s", 32, instrNames[vIns2]);
                      } else {
                                 ins2.setTxt("--");
                      }

                      if (opTypeRs1 == OP_TYPE_REG)
vRs1 = (vRs1 % 16);

if (opTypeRs2 == OP_TYPE_REG)
vRs2 = (vRs2 % 16);

if (opTypeRs3 == OP_TYPE_REG)
vRs3 = (vRs3 % 16);


if (opTypeRdt == OP_TYPE_UNUSED) rdt.setTxt("-") else rdt.setTxt("R%d", vRdt);

if (opTypeRs1 == OP_TYPE_UNUSED){
                                 rs1.setTxt("-");
                      } else if (opTypeRs1 == OP_TYPE_IMM) {
                                 rs1.setTxt("%02X", vRs1);
                      } else {
                                 if(vIns == LDR || vIns == STR || vIns == LDRi || vIns == STRi) {
                                            rs1.setTxt("[R%d", vRs1)
                                 } else {
                                            rs1.setTxt("R%d", vRs1);
                                 }
                      }

if (opTypeRs2 == OP_TYPE_UNUSED)
rs2.setTxt("-")
else if (opTypeRs2 == OP_TYPE_REG) {
                                 if((vIns == LDR || vIns == STR) && opTypeRs3 == OP_TYPE_UNUSED) {
                                            rs2.setTxt("R%d]", vRs2);
                                 } else {
                                            rs2.setTxt("R%d", vRs2);
                                 }
} else {
                                 if((vIns == LDRi || vIns == STRi) && opTypeRs3 == OP_TYPE_UNUSED) {
                                            rs2.setTxt("%02X]", vRs2);
                                 } else {
                                            rs2.setTxt("%02X", vRs2);
                                 }
                      }

if (opTypeRs3 == OP_TYPE_UNUSED)
rs3.setTxt("-")
else if (opTypeRs3 == OP_TYPE_REG){
if(vIns == LDR || vIns == STR) {
                                            rs3.setTxt("R%d]", vRs3)
                                 } else {
                                            rs3.setTxt("R%d", vRs3)
                                 }
} else {
if(vIns == LDR || vIns == STR) {
                                            rs3.setTxt("%02X]", vRs3);
                                 } else {
                                            rs3.setTxt("%02X", vRs3)
                                 }
                      }

if (instrIsBranch(vIns) ) {
if (vRs2 & 0x80) {
// Jump up
offset = (se8(vRs2)/4)*h + h/2;                                              // relative
arrowUp.setPt(2, x - 7, y + offset);
arrowUp.setPt(3, x - 2, y + offset);
arrowUp.setOpacity(1);
arrowDown.setOpacity(0);
} else {
// Jump down
offset = (vRs2/4)*h + h/2;                                                              // relative jump
arrowDown.setPt(2, x + w + 7, y + offset);
arrowDown.setPt(3, x + w + 2, y + offset);
arrowDown.setOpacity(1);
arrowUp.setOpacity(0);
}
} else {
arrowUp.setOpacity(0);
arrowDown.setOpacity(0);
}

if (remember) {
string s = sprintf("i%d", addr / 4);
                                 setArg(s, getOpcode().toString());

example = 0;
                                 setArg("example", example.toString());
}

}

function setValue(num instr, string sBit, num cond, num rdt, num rs1, num rs2imm, num instr2, num rs3imm) {
vIns = instr;
                      vSBit = sBit;
                      vCond = cond;
vRdt = rdt;
vRs1 = rs1 & 0xff;
vRs2 = rs2imm & 0xff;
                      vIns2 = instr2;
                      vRs3 = rs3imm;
initRegs(0);
}

function setOpcode(num opcode) {
vIns = (opcode & 0xFF000000) >> 24;
vRdt = (opcode & 0x00FF0000) >> 16;
vRs1 = (opcode & 0x0000FF00) >> 8;
vRs2 = (opcode & 0x000000FF);
initRegs(0);
}

when ins ~> eventMB(num down, num flags, num x, num y) {
if (!lockCircuit) {
if (down) {
clk = timeMS();
                                            if (flags & MB_LEFT) {
           vIns = (vIns == MAX_INSTR) ? 0 : vIns + 1;
                                            } else if (flags & MB_RIGHT) {
                                                       vIns = (vIns == 0) ? MAX_INSTR : vIns - 1;
                                            }
} else {
clk = clk + 500;
if (timeMS() > clk)
vIns = 0;
}
initRegs(1);
}
                      return 0;
}
```

```
    when sBit ~> eventMB(num down, num flags, num x, num y) {
if (!lockCircuit && sOk) {
if (down) {
clk = timeMS();
                if (flags & MB_LEFT) {
    vSBit = (vSBit == " ") ? "S" : " ";
                } else if (flags & MB_RIGHT) {
    vSBit = (vSBit == " ") ? "S" : " ";
                }
} else {
clk = clk + 500;
if (timeMS() > clk)
vSBit = 0;
}
initRegs(1);
}
        return 0;
}


when cond ~> eventMB(num down, num flags, num x, num y) {
if (!lockCircuit) {
if (down) {
clk = timeMS();
                if (flags & MB_LEFT) {
    vCond = (vCond == MAX_COND) ? 0 : vCond + 1;
                } else if (flags & MB_RIGHT) {
                    vCond = (vCond == 0) ? MAX_COND : vCond - 1;
                }
} else {
clk = clk + 500;
if (timeMS() > clk)
vCond = 0;
}
initRegs(1);
}
        return 0;
}

when ins2 ~> eventMB(num down, num flags, num x, num y) {
if (!lockCircuit && shiftOk) {
if (down) {
clk = timeMS();
                //Special functionallity so only shift operations possible
                if (flags & MB_LEFT) {
                    if(vIns2 == 0) { //Jump to shift instructions
                        vIns2 = LSL;
                    } else if(vIns2 == RORi) { //loop back to start
                        vIns2 = 0;
                    } else if (vIns2 == ROR) { //Jump to immediate shift ops
                        vIns2 = LSLi;
                    } else {
                        vIns2 = vIns2 + 1;
                    }
                } else if (flags & MB_RIGHT) {
                    if(vIns2 == 0) {
                        vIns2 = RORi;
                    } else if(vIns2 == LSL) { //loop back to start
                        vIns2 = 0;
                    } else if (vIns2 == LSLi) { //Jump to immediate shift ops
                        vIns2 = ROR;
                    } else {
                        vIns2 = vIns2 - 1;
                    }
                }
} else {
clk = clk + 500;
if (timeMS() > clk)
vIns2 = 0;
}
initRegs(1);
}
        return 0;
}

when rdt ~> eventMB(num down, num flags, num x, num y) {
if (!lockCircuit && down && opTypeRdt != OP_TYPE_UNUSED) {
if (flags & MB_LEFT) {
                vRdt = (vRdt == 15) ? 0 : vRdt + 1;
} else if (flags & MB_RIGHT)
                vRdt = (vRdt == 0) ? 15 : vRdt - 1;
```

```
            initRegs(1);
        }
        return 0;
}

    when rs1 ~> eventMB(num down, num flags, num x, num y) {
if (!lockCircuit && down) {
if (flags & MB_LEFT) {
if (opTypeRs1 == OP_TYPE_REG) {
vRs1 = (vRs1 + 1) % 16;
} else if (opTypeRs1 == OP_TYPE_IMM) {
clk = timeMS();
vRs1 = (vRs1 + 1) % 256;
}
            } else if (flags & MB_RIGHT) {
                if (opTypeRs1 == OP_TYPE_REG) {
vRs1 = (vRs1 - 1) % 16;
if (vRs1 < 0)
vRs1 = 16 + vRs1;
} else if (opTypeRs1 == OP_TYPE_IMM) {
clk = timeMS();
vRs1 = (vRs1 - 1) % 256;
if (vRs1 < 0)
vRs1 = 256 + vRs1;
}
} else {
if (opTypeRs1 == OP_TYPE_IMM) {
clk = clk + 500;
if (timeMS() > clk)
vRs1 = 0;
}
}
            initRegs(1);
        }
        return 0;
}

    when rs2 ~> eventMB(num down, num flags, num x, num y) {
if (!lockCircuit && down) {
if (flags & MB_LEFT) {
if (opTypeRs2 == OP_TYPE_REG) {
vRs2 = (vRs2 + 1) % 16;
} else if (opTypeRs2 == OP_TYPE_IMM) {
clk = timeMS();
vRs2 = (vRs2 + 1) % 256;
}
            } else if (flags & MB_RIGHT) {
                if (opTypeRs2 == OP_TYPE_REG) {
vRs2 = (vRs2 - 1) % 16;
if (vRs2 < 0)
vRs2 = 16 + vRs2;
} else if (opTypeRs2 == OP_TYPE_IMM) {
clk = timeMS();
vRs2 = (vRs2 - 1) % 256;
if (vRs2 < 0)
vRs2 = 256 + vRs2;
}
} else {
if (opTypeRs2 == OP_TYPE_IMM) {
clk = clk + 500;
if (timeMS() > clk)
vRs2 = 0;
}
}
initRegs(1);
}
}

    when rs3 ~> eventMB(num down, num flags, num x, num y) {
if (!lockCircuit && down) {
if (flags & MB_LEFT) {
if (opTypeRs3 == OP_TYPE_REG) {
vRs3 = (vRs3 + 1) %16;
} else if (opTypeRs3 == OP_TYPE_IMM) {
clk = timeMS();
vRs3 = (vRs3 + 1) % 256;
}
            } else if (flags & MB_RIGHT) {
                if (opTypeRs3 == OP_TYPE_REG) {
vRs3 = (vRs3 - 1) % 16;
if (vRs3 < 0)
vRs3 = 16 + vRs3;
} else if (opTypeRs3 == OP_TYPE_IMM) {
```

```
clk = timeMS();
vRs3 = (vRs3 - 1) % 256;
if (vRs3 < 0)
vRs3 = 256 + vRs3;
}
} else {
if (opTypeRs3 == OP_TYPE_IMM) {
clk = clk + 500;
if (timeMS() > clk)
vRs3 = 0;
}
}
initRegs(1);
}
}

}

//
// Instruction Memory
//
class InstructionMemory(num x, num y, num w, num h) {


           //Headings
           Pen headingsPen = SolidPen(0, 0, RED);
           Font headingsFont = Font("Calibri", 18);

           Rectangle addrLabel = Rectangle2(0, 0, 0, 0, x+13, y, 10, 4, headingsPen, headingsFont, "Addr");
           Rectangle instrLabel = Rectangle2(0, 0, 0, 0 ,47 + x+13, y, 10, 4, headingsPen, headingsFont, "Instr");
           Rectangle sBitLabel = Rectangle2(0, 0, 0, 0 ,99 + x+13, y, 10, 4, headingsPen, headingsFont, "S");
           Rectangle condLabel = Rectangle2(0, 0, 0, 0 ,130 + x+13, y, 10, 4, headingsPen, headingsFont, "Cond");
           Rectangle rdLabel = Rectangle2(0, 0, 0, 0 ,175 + x+13, y, 10, 4, headingsPen, headingsFont, "Rd");
           Rectangle op1Label = Rectangle2(0, 0, 0, 0 ,210 + x+13, y, 10, 4, headingsPen, headingsFont, "Op1");
           Rectangle op2Label = Rectangle2(0, 0, 0, 0 ,250 + x+13, y, 10, 4, headingsPen, headingsFont, "Op2");
           Rectangle offsetLabel = Rectangle2(0, 0, 0, 0 ,300 + x+13, y, 10, 4, headingsPen, headingsFont, "Offset");

num ih = (h-12 - 4) / 32;
Instruction instruction[32];
num active = 31;

Rectangle r = Rectangle2(0, 0, blackPen, instrMemBrush, x, y+12, w, h-12);
           r.setRounded(2, 2);
Rectangle2(0, 0, blackPen, instrMemValueBrush, x + 2, y+12 + 2, w - 4, h-12 - 4);

for (num lp1 = 0; lp1 < 32; lp1++)
           instruction[lp1] = Instruction(x + 2, y+12 + 2 + lp1 * ih, w - 4, ih, lp1 * 4);

function setValue(num addr, num instr, string sBit, num cond, num rdt, num rs1, num rs2imm, num instr2, num rs3imm) {
instruction[addr / 4].setValue(instr, sBit, cond, rdt, rs1, rs2imm, instr2, rs3imm);
}

num function getOpcode(num addr) {
return instruction[addr / 4].getOpcode();
}

function setOpcode(num addr, num opcode) {
instruction[addr / 4].setOpcode(opcode);
}

function setActive(num addr) {
instruction[active].dot.setOpacity(0);
active = addr / 4;
instruction[active].dot.setOpacity(1);
}

}

// eof

//
// instructionregister.vin
//
// Simulation of the ARM
// Written by Rónán Dowling-Cullen, Final Year Computer Science 2018/19DLX
// Building on the simulation of the DLX written by Edsko de Vries, Summer 2003

class InstructionRegister(num x, num y, num w, num h, string caption) {

           string vSBit = " ";
num vIns = EMPTY, vCond = 0, vRdt = 0, vRs1 = 0, vRs2 = 0, vIns2 = EMPTY, vRs3 = 0;
           string nSBit = " ";
num nIns = EMPTY, nCond = 0, nRdt = 0, nRs1 = 0, nRs2 = 0, nIns2 = EMPTY, nRs3 = 0;
```

```
           Font IMFont = Font("Calibri",18);

string txt = "EMPTY";

Rectangle r1 = Rectangle2(0, 0, blackPen, blackBrush, x, y, w, h);
           r1.setRounded(2, 2);
Rectangle r2 = Rectangle2(0, 0, blackPen, whiteBrush, x + 5, y + 4, w - 11, h - 24);
           r2.setRounded(2, 2);
Rectangle r3 = Rectangle2(0, 0, 0, 0, x, y + h - 20, w, 22, whitePen, font, caption);
Rectangle label = Txt(valuesLayer, 0, x + w/2, y + (h - 24)/2 + 4, 0, IMFont, txt);
           label.rotate(-90);

//
// setNewValue
//
function setNewValue(num instr, string sBit, num cond, num rdt, num rs1, num rs2, num instr2, num rs3) {
nIns = instr;
                      nSBit = sBit;
                      nCond = cond;
nRdt = rdt;
nRs1 = rs1;
nRs2 = rs2;
nIns2 = instr2;
nRs3 = rs3;
}

//
// setNewInstruction
//
function setNewInstruction(Instruction i) {
nIns = i.vIns;
                      nSBit = i.vSBit;
                      nCond = i.vCond;
nRdt = i.vRdt;
nRs1 = i.vRs1;
nRs2 = i.vRs2;
nIns2 = i.vIns2;
nRs3 = i.vRs3;
}

//
// getNewInstrTxt
//
string function getNewInstrTxt() {
return instrText(nIns, nSBit, nCond, nRdt, nRs1, nRs2, nIns2, nRs3);
}

//
// update
//
function update() {
vIns = nIns;
                      vSBit = nSBit;
                      vCond = nCond;
vRdt = nRdt;
vRs1 = nRs1;
vRs2 = nRs2;
vIns2 = nIns2;
                      vRs3 = nRs3;
txt = instrText(vIns, vSBit, vCond, vRdt, vRs1, vRs2, nIns2, nRs3);
label.setTxt(txt);
r2.setBrush(yellowBrush);
wait(16);
r2.setBrush(whiteBrush);
}

//
// setOpacity
//
function setOpacity(num opacity) {
r1.setOpacity(opacity);
r2.setOpacity(opacity);
r3.setOpacity(opacity);
label.setOpacity(opacity);
}

           //
// reset
//
function reset() {
vIns = EMPTY;
                      vIns2 = EMPTY;
```

```
                       vSBit = " ";
vRdt = vCond = vRs1 = vRs2 = vRs3 = 0;
nIns = EMPTY;
                       nIns2 = EMPTY;
                       nSBit = " ";
nRdt = nRs1 = nRs2 = nRs3 = 0;
txt = instrText(vIns, vSBit, vCond, vRdt, vRs1, vRs2, nIns2, nRs3);
label.setTxt(txt);
}

}

// eof

//
// instructions.vin
//
// Simulation of the ARM
// Written by Rónán Dowling-Cullen, Final Year Computer Science 2018/19DLX
// Building on the simulation of the DLX written by Edsko de Vries, Summer 2003


//
// Condition Constants
//

// Number of Conditions
const num MAX_COND = 18;

const num def = 0;    // Default is equivalent to AL
const num EQ = 1;                // Z set (Equal)
const num NE = 2;                // Z clear (Not Equal)
const num HS = 3;                // C set (Unsigned Higher or same)
const num CS = 4;                // C set
const num LO = 5;                // C clear (Unsigned Lower or Same)
const num CC = 6;                // C clear
const num MI = 7;                // N set (Negative)
const num PL = 8;                // N clear (Positive or Zero)
const num VS = 9;                // V set (Overflow)
const num VC = 10;               // V clear (No Overflow)
const num HI = 11;               // C set and Z clear (Unsigned Higher)
const num LS = 12;               // C clear or Z set (Unsigned Lower or Same)
const num GE = 13;               // N set and V set, or N clear and V clear (Greater Than or Equal)
const num LT = 14;               // N set and V clear, or N clear and V set(Less Than)
const num GT = 15;               // Z clear, and either N set and V set, or N clear and V clear
const num LE = 16;               // Z set, or N set and V clear, or N clear and V set (Equal)
const num AL = 17;               // (Always)
const num NV = 18;               // (Reserved)

//
// Condition Names
//

string condNames[MAX_COND+1];


condNames[def]= " ";  // Default is equivalent to AL
condNames[EQ] = "EQ";                      // Z set (Equal)
condNames[NE] = "NE";                      // Z clear (Not Equal)
condNames[HS] = "HS";                      // C set (Unsigned Higher or same)
condNames[CS] = "CS";                      // C set
condNames[LO] = "LO";                      // C clear (Unsigned Lower or Same)
condNames[CC] = "CC";                      // C clear
condNames[MI] = "MI";                      // N set (Negative)
condNames[PL] = "PL";                      // N clear (Positive or Zero)
condNames[VS] = "VS";                      // V set (Overflow)
condNames[VC] = "VC";                      // V clear (No Overflow)
condNames[HI] = "HI";                      // C set and Z clear (Unsigned Higher)
condNames[LS] = "LS";                      // C clear or Z set (Unsigned Lower or Same)
condNames[GE] = "GE";                      // N set and V set, or N clear and V clear (Greater Than or Equal)
condNames[LT] = "LT";                      // N set and V clear, or N clear and V set(Less Than)
condNames[GT] = "GT";                      // Z clear, and either N set and V set, or N clear and V clear
condNames[LE] = "LE";                      // Z set, or N set and V clear, or N clear and V set (Equal)
condNames[AL] = "AL";                      // (Always)
condNames[NV] = "NV";                      // (Reserved)


//
// Instruction constants
//

// Number of instructions (excluding "stall" instruction)
const num MAX_INSTR = 48;
```

```
// No-op
const num NOP = 0;

// Arithmetic/Logical/Shifting instructions (register-register)

const num ADD = 1; // Add
const num ADC         = 2;       // Add with Carry
const num SUB = 3; // Subtract
const num SBC         = 4;       // Subtract with Carry
const num RSB         = 5;       // Reverse Subtract
const num RSC         = 6;       // Reverse Subtract with Carry
const num MUL         = 7;       // Multiply

const num AND = 8; // Logical AND
const num ORR = 9; // Logical OR
const num EOR         = 10;      // Exclusive OR
const num BIC         = 11;      // Bit Clear

const num LSL         = 12;      // Logical Shift Left
const num LSR         = 13;      // Logical Shift Right
const num ASR         = 14;      // Arithmetic Shift RIght
const num ROR         = 15;      // Rotate Right


// Arithmetic/Logical/Shifting instructions (register-immediate)


const num ADDi = 16; // Add
const num ADCi        = 17;      // Add with Carry
const num SUBi = 18; // Subtract
const num SBCi        = 19;      // Subtract with Carry
const num RSBi        = 20;      // Reverse Subtract
const num RSCi        = 21;      // Reverse Subtract with Carry

const num ANDi = 22; // Logical AND
const num ORRi = 23; // Logical OR
const num EORi        = 24;      // Exclusive OR
const num BICi        = 25;      // Bit Clear

const num LSLi        = 26;      // Logical Shift Left
const num LSRi        = 27;      // Logical Shift Right
const num ASRi        = 28;      // Arithmetic Shift RIght
const num RORi        = 29;      // Rotate Right


// Comparisons (register - register)
const num CMN         = 30;      // Compare Negative
const num CMP         = 31;      // Compare
const num TEQ         = 32;      // Test bitwise equality
const num TST         = 33;      // Test bits

// Comparisons (register - immediate)
const num CMNi        = 34;      // Compare Negative
const num CMPi        = 35;      // Compare
const num TEQi        = 36;      // Test bitwise equality
const num TSTi        = 37;      // Test bits



// Load/store instructions (register - register)
const num MOV         = 38;      // Move register or constant
const num MVN         = 39;      // Move negative register
const num LDR         = 40; // Load register from memory
const num STR         = 41; // Store register to memory

// Load/store instructions (register - immediate)
const num MOVi        = 42;      // Move register or constant
const num MVNi        = 43;      // Move negative register
const num LDRi        = 44; // Load register from memory
const num STRi        = 45; // Store register to memory


// Control instructions (register - immediate)
const num B                      = 46; // Branch
const num BL          = 47; // Branch with Link


// Halt
const num HALT = 48; // Stop running

// "Stall" and "Empty" instruction (aliases for NOP)
const num STALL = 49;
const num EMPTY = 50;

//
// Instruction names
//

string instrNames[51];

// NOP
instrNames[NOP] = "NOP";

instrNames[ADD] = "ADD"; // Add
instrNames[ADC]       = "ADC";   // Add with Carry
instrNames[SUB] = "SUB"; // Subtract
instrNames[SBC]       = "SBC";   // Subtract with Carry
instrNames[RSB]       = "RSB";   // Reverse Subtract
instrNames[RSC]       = "RSC";   // Reverse Subtract with Carry
instrNames[MUL]       = "MUL";   // Multiply

instrNames[AND] = "AND"; // Logical AND
instrNames[ORR] = "ORR"; // Logical OR
instrNames[EOR]       = "EOR";   // Exclusive OR
instrNames[BIC]       = "BIC";   // Bit Clear
instrNames[LSL]       = "LSL";   // Logical Shift Left
instrNames[LSR]       = "LSR";   // Logical Shift Right
instrNames[ASR]       = "ASR";   // Arithmetic Shift RIght
instrNames[ROR]       = "ROR";   // Rotate Right


// Arithmetic/Logical/Shifting instructions (register-immediate)


instrNames[ADDi] = "ADD(i)"; // Add
instrNames[ADCi] = "ADC(i)";     // Add with Carry
instrNames[SUBi] = "SUB(i)"; // Subtract
instrNames[SBCi] = "SBC(i)";     // Subtract with Carry
instrNames[RSBi] = "RSB(i)";     // Reverse Subtract
instrNames[RSCi] = "RSC(i)";     // Reverse Subtract with Carry
instrNames[ANDi] = "AND(i)"; // Logical AND
instrNames[ORRi] = "ORR(i)"; // Logical OR
instrNames[EORi] = "EOR(i)";     // Exclusive OR
instrNames[BICi] = "BIC(i)";     // Bit Clear

instrNames[LSLi] = "LSL(i)";     // Logical Shift Left
instrNames[LSRi] = "LSR(i)";     // Logical Shift Right
instrNames[ASRi] = "ASR(i)";     // Arithmetic Shift RIght
instrNames[RORi] = "ROR(i)";     // Rotate Right


// Comparisons (register - register)
instrNames[CMN]       = "CMN";   // Compare Negative
instrNames[CMP]       = "CMP";   // Compare
instrNames[TEQ]       = "TEQ";   // Test bitwise equality
instrNames[TST]       = "TST";   // Test bits

// Comparisons (register - immediate)
instrNames[CMNi]      = "CMN(i)";           // Compare Negative
instrNames[CMPi]      = "CMP(i)";           // Compare
instrNames[TEQi]      = "TEQ(i)";           // Test bitwise equality
instrNames[TSTi]      = "TST(i)";           // Test bits


// Load/store instructions (register - register)
instrNames[MOV]       = "MOV";   // Move register or constant
instrNames[MVN]       = "MVN";   // Move negative register
instrNames[LDR]       = "LDR"; // Load register from memory
instrNames[STR]       = "STR"; // Store register to memory

// Load/store instructions (register - immediate)
instrNames[MOVi]      = "MOV(i)";           // Move register or constant
instrNames[MVNi]      = "MVN(i)";           // Move negative register
instrNames[LDRi]      = "LDR(i)"; // Load register from memory
instrNames[STRi]      = "STR(i)"; // Store register to memory


// Control instructions (register - register)
instrNames[B]         = "B"; // Branch
instrNames[BL]        = "BL"; // Branch with Link

// Halt
instrNames[HALT] = "HALT";

// "Stall" instruction
instrNames[STALL] = "STALL";
instrNames[EMPTY] = "EMPTY";

//
// Instruction classes
//
num function instrIsNop(num instr) {
return (instr == NOP || instr == STALL || instr == EMPTY || instr == HALT) ? 1 : 0;
}

num function instrIsArRR(num instr) {
           return (instr >= ADD && instr <= ROR) ? 1 : 0;
}

num function instrIsArRI(num instr) {
return ((instr >= ADDi) && (instr <= RORi)) ? 1 : 0;
           return 0;
}

num function instrIsBranch(num instr) {
           return (instr == B || instr == BL) ? 1 : 0;
}


num function instrIsLoadOrStore(num instr) {
return ((instr == LDR) || (instr == STR) || (instr == LDRi) || (instr == STRi)) ? 1 : 0;
}

// Shift immediate

num function instrIsShI(num instr) {
           return (instr >= LSLi && instr <= RORi) ? 1 : 0;
}

// instructions with no destination register Register - Register
num function instrHasNoDstRR(num instr) {
           return ((instr >= CMN && instr <= TST)|| instrIsBranch(instr)) ? 1 : 0;
}

// instructions with no destination register Register - Immediate
num function instrHasNoDstRI(num instr) {
           return ((instr >= CMNi && instr <= TSTi) || instrIsBranch(instr)) ? 1 : 0;
}

// instructions with no op2 Register - Register
num function instrHasNoOp1RR(num instr) {
           return ((instr >= MOV && instr <= MVN)) ? 1 : 0;
}

// instructions with no op2 Register - Immediate
num function instrHasNoOp1RI(num instr) {
           return ((instr >= MOVi && instr <= MVNi)) ? 1 : 0;
}


//
// Instruction operand types
//
const num OP_TYPE_UNUSED = 0;
const num OP_TYPE_REG = 1;
const num OP_TYPE_IMM = 2;

num function instrOpTypeRdt(num instr) {
if (instrIsNop(instr) || instrHasNoDstRR(instr) || instrHasNoDstRI(instr))
return OP_TYPE_UNUSED;
else
return OP_TYPE_REG;
}

num function instrOpTypeRs1(num instr) {
if (instrIsNop(instr) || instrHasNoOp1RR(instr) || instrHasNoOp1RI(instr) || instrIsBranch(instr))
return OP_TYPE_UNUSED;
else
return OP_TYPE_REG;
}

num function instrOpTypeRs2(num instr) {
if (instrIsNop(instr))
return OP_TYPE_UNUSED;
else if ((instrIsArRR(instr) || instrHasNoDstRR(instr) || (instr == LDR) || (instr == STR) || instrHasNoOp1RR(instr)) &&
!instrIsBranch(instr))
return OP_TYPE_REG;
else
```

```
return OP_TYPE_IMM;
}

num function instrOpTypeRs3(num instr, num instr2) {
if (instrIsNop(instr2) || instrIsNop(instr) || instrIsArRI(instr) || (instr >= LSL && instr <= ROR) || (instr >= CMNi && instr <=
TSTi) || (instr >= MOVi))
return OP_TYPE_UNUSED;
else if (instrIsShI(instr2))
return OP_TYPE_IMM;
else
return OP_TYPE_REG;
}

num function shiftAllowed(num instr){
        if (instrIsNop(instr) || instrIsArRI(instr) || (instr >= LSL && instr <= ROR) || (instr >= CMNi && instr <= TSTi) ||
(instr >= MOVi))
                return 0;
        return 1;
}

num function sAllowed(num instr){
        if(instrIsNop(instr) || instrIsLoadOrStore(instr) || instrIsBranch(instr))
                return 0;
        return 1;
}

//
// Instruction formatting
//
string function instrText(num instr, string s, num cond, num rdt, num rs1, num rs2, num instr2, num rs3) {
if (instrIsNop(instr))
return sprintf("%s", instrNames[instr]);
else if (instr >= ADD && instr <= BIC)
return sprintf("%s%s%s R%d,R%d,R%d,%s R%d", instrNames[instr], s, condNames[cond], rdt, rs1, rs2, ((instr2 != NOP)?
instrNames[instr2]:""), rs3);
else if (instr >= LSL && instr <= ROR)
return sprintf("MOV%s%s R%d,R%d,%s R%d", s, condNames[cond], rdt, rs1, instrNames[instr], rs2);
else if (instr >= ADDi && instr <= BICi)
return sprintf("%s%s%s R%d,R%d,%02X", instrNames[instr], s, condNames[cond], rdt, rs1, rs2);
else if (instr >= LSLi && instr <= RORi)
return sprintf("MOV%s%s R%d,R%d,%s %02X", s, condNames[cond], rdt, rs1, instrNames[instr], rs2);
else if (instr >= CMN && instr <= TST)
return sprintf("%s%s%s R%d,R%d,R%d,%s R%d", instrNames[instr], s, condNames[cond], rdt, rs1, rs2, ((instr2 != NOP)?
instrNames[instr2]:""), rs3);
else if (instr >= CMNi && instr <= TSTi)
return sprintf("%s%s%s R%d,R%d,%02X", instrNames[instr], s, condNames[cond], rdt, rs1, rs2);
else if (instr == LDR || instr == STR)

return sprintf("%s%s R%d,[R%d,R%d,%s R%d]", instrNames[instr], condNames[cond], rdt, rs1, rs2, ((instr2 != NOP)?
instrNames[instr2]:""), rs3);
else if (instr == LDRi || instr == STRi)

return sprintf("%s%s R%d,[R%d,%02X]", instrNames[instr], condNames[cond], rdt, rs1, rs2);
else if (instr == MOV || instr == MVN)

return sprintf("%s%s%s R%d,R%d,%s R%d", instrNames[instr], s, condNames[cond], rdt, rs1, ((instr2 != NOP)? instrNames[instr2]:""),
rs3);
else if (instr == MOVi || instr == MVNi)
return sprintf("%s%s%s R%d,%02X", instrNames[instr], s, condNames[cond], rdt, rs1);
else if (instrIsBranch(instr))
return sprintf("%s%s %02X", instrNames[instr], condNames[cond], rs2);
return "EMPTY";
}

//
// Execution
//

// 8-bit sign extension
num function se8(num t) {
if (t & 0x80)
return (-1 ^ 0xFF | t);
else
return t;
}


// Check Condition Codes
num function isN(num val){
        if(val & 0x80)
                return 1;
        else
                return 0;
```

```
}

num function isZ(num val){
        if(val == 0)
                return 1;
        else
                return 0;
}

num function isAddC(num op1, num op2){
        if((op1 + op2) & 0x100)
                return 1;
        else
                return 0;
}

num function isSubC(num op1, num op2){
        if((op1 + op2) & 0x100)
                return 0;
        else
                return 1;
}

num function isAddV(num op1, num op2, num res){
        if(((op1 & 0x80) == (op2 & 0x80)) && ((res & 0x80) != (op1 & 0x80))) {
                return 1;
        } else {
                return 0;
        }
}

num function isSubV(num op1, num op2, num res){
        if(((op1 & 0x80) != (op2 & 0x80)) && ((res & 0x80) != (op1 & 0x80)))
                return 1;
        else
                return 0;
}


//Function to set flags based on res, op1 and op2
num function setFlags(num res, num op1, num op2, num add, num mul) {
        cpsr0.setNNValue(isN(res&0xFF));
        cpsr0.setNZValue(isZ(res&0xFF));
        if(add){
                cpsr0.setNCValue(isAddC(se8(op1),se8(op2)));
                if(!mul)
                        cpsr0.setNVValue(isAddV(se8(op1),se8(op2),(res&0xFF)));
        } else {
                cpsr0.setNCValue(isSubC(se8(op1),se8(op2)));
                if(!mul)
                        cpsr0.setNVValue(isSubV(se8(op1),se8(op2),(res&0xFF)));
        }
}

//Test Condition On Function
num function testCondition(num cond) {

        CPSR tempCPSR;
        if(!cpsr0.invalid){
                tempCPSR = cpsr0;
        } else if (!cpsr1.invalid){
                tempCPSR = cpsr1;
        } else {
                tempCPSR = cpsr;
        }


        if(cond == def || cond == AL) {  //Always
                return 1;
        } else if (cond == EQ) {                    // Z set (cond) {= Equal)
                return(tempCPSR.z);
        } else if (cond == NE) {                    // Z clear (cond) {= Not Equal)
                return(!tempCPSR.z);
        } else if (cond == HS) {                    // C set (cond) {= Unsigned Higher or same)
                return(tempCPSR.c);
        } else if (cond == CS) {                    // C set
                return(tempCPSR.c);
        } else if (cond == LO) {                    // C clear (cond) {= Unsigned Lower or Same)
                return(!tempCPSR.c);
        } else if (cond == CC) {                    // C clear
                return(!tempCPSR.c);
        } else if (cond == MI) {                    // N set (cond) {= Negative)
                return(tempCPSR.n);
        } else if (cond == PL) {                        // N clear (cond) {= Positive or Zero)
                return(!tempCPSR.n);
        } else if (cond == VS) {                        // V set (cond) {= Overflow)
                return(tempCPSR.v);
        } else if (cond == VC) {                        // V clear (cond) {= No Overflow)
                return(!tempCPSR.v);
        } else if (cond == HI) {                        // C set and Z clear (cond) {= Unsigned Higher)
                return(tempCPSR.c && !tempCPSR.z);
        } else if (cond == LS) {                        // C clear or Z set (cond) {= Unsigned Lower or Same)
                return(!tempCPSR.c || tempCPSR.z);
        } else if (cond == GE) {                        // N set and V set, or N clear and V clear (cond) {= Greater Than or Equal)
                return((tempCPSR.n && tempCPSR.v) || (!tempCPSR.n && !tempCPSR.v));
        } else if (cond == LT) {                        // N set and V clear, or N clear and V set(cond) {= Less Than)
                return((tempCPSR.n && !tempCPSR.v) || (!tempCPSR.n && tempCPSR.v));
        } else if (cond == GT) {                        // Z clear, and either N set and V set, or N clear and V clear
                return(!tempCPSR.z && ((tempCPSR.n && tempCPSR.v) || (!tempCPSR.n && !tempCPSR.v)));
        } else if (cond == LE) {                        // Z set, or N set and V clear, or N clear and V set (cond) {= Equal)
                return(tempCPSR.z || (tempCPSR.n && !tempCPSR.v) || (!tempCPSR.n && tempCPSR.v));
        } else if (cond == NV) {                        // (cond) {= Reserved)
                return 0;
        }
}

num function instrExecute(num instr, string s, num cond, num op1, num op2, num shiftIns, num op3, num n, num z, num c, num v) {

        //Perform Shift on OP2
        if(shiftIns == LSL || shiftIns == LSLi){
                op2 = (op2 << op3) & 0xFF;
        } else if(shiftIns == LSR || shiftIns == LSRi){
                op2 = (op2 >> op3) & 0xFF;
        } else if(shiftIns == ASR || shiftIns == ASRi){
                num sign = op2 & 0x80;
                op2 = (op2 >> op3) & 0xFF;
                if(sign){
                        op2 = (op2 | (0xFF << (8-op3))) & 0xFF;
                }
        } else if(shiftIns == ROR || shiftIns == RORi){
                op3 = op3%8;
                op2 = ((op2 >> op3) | (op2 << (8-op3))) & 0xFF;
        }

        num res = 0;                                               // Init result to zero
        num add = 0;                                               // Init is Add instruction      to zero
        num mul = 0;                                               // Init is Mul instruction      to zero
        num sOK = 1;                                               // Is S allowed on this instruction

if (instr == ADD || instr == ADDi) {
res = (se8(op1) + se8(op2)) & 0xFF;
                add = 1;
} else if (instr == SUB || instr == SUBi) {
res = (se8(op1) - se8(op2)) & 0xFF;
} else if (instr == AND || instr == ANDi) {
res = op1 & op2;
} else if (instr == ORR || instr == ORRi) {
res = op1 | op2;
} else if (instr == EOR || instr == EORi) {
res = op1 ^ op2;


} else if (instr == LSL || instr == LSLi) {
res = (op1 << op2) & 0xFF;
} else if (instr == LSR || instr == LSRi) {
res = (op1 >> op2) & 0xFF;
        } else if (instr == ROR || instr == RORi) {
                op2 = op2 % 8;                  //modulo by max shift to remove need for loop
                num newLeftPart = (op1 << (8 - op2)) & 0xFF;
                num newRightPart = (op1 >> op2) & 0xFF;
                res = newLeftPart + newRightPart;
} else if (instr == ASR || instr == ASRi) {
                num neg = op1 >> 7;
                if(op2 > 8) op2 = 8;  //If shifting by more than register size
                if(neg){
                        op1 = op1 ^ 0xFF00;             //introduce ones before if negative to be shifted into place
                }
                res = (op1 >> op2) & 0xFF;


        } else if (instr == LDR || instr == STR || instr == LDRi || instr == STRi) {
res = (se8(op1) + se8(op2)) & 0xFF;
                add = 1;
                sOK = 0;
        } else if (instr == ADC || instr == ADCi) {
res = (se8(op1) + se8(op2) + c) & 0xFF;
                add = 1;
        } else if (instr == BIC || instr == BICi) {
                res = (op1 & (~op2));
        } else if (instr == CMN || instr == CMNi) {
                res = se8(op1) + se8(op2);
                add = 1;
                setFlags(res,op1,op2,add,0);
                res = res & 0xFF;
        }
        else if (instr == CMP || instr == CMPi) {
                res = se8(op1) - se8(op2);
                setFlags(res,op1,op2,add,0);
                res = res & 0xFF;
        }
        else if (instr == MOV || instr == MOVi) {
                res =((op2) & 0xFF);
        }
        else if (instr == MUL) {
                res =((op1 * op2) & 0xFF);
                mul = 1;   //If S set V preserved on MUL
        }
        else if (instr == MVN || instr == MVNi) {
                res =((~op1) & 0xFF);
        }
        else if (instr == RSB || instr == RSBi) {
                res = (se8(op2) - se8(op1)) & 0xFF;
        } else if (instr == RSC || instr == RSCi) {
                res = (se8(op2) - se8(op1)-1+c) & 0xFF;
        } else if (instr == SBC || instr == SBCi) {
                res = (se8(op1) - se8(op2)-1+c) & 0xFF;
        }
        else if (instr == TEQ || instr == TEQi) {
                res = se8(op1) ^ se8(op2);
                setFlags(res,op1,op2,add,0);
                res = res & 0xFF;
        }
        else if (instr == TST || instr == TSTi) {
                res = se8(op1) & se8(op2);
                setFlags(res,op1,op2,add,0);
                res = res & 0xFF;
        }
else if (instr == BL) {
res = op2 & 0xFF;
                sOK = 0;
} else {
res = 0xEE;
        }


        // If S set, set CPSR
        if(s == "S" && sOK){
                setFlags(res,op1,op2,add,mul);
        }

        return res;
}

// eof

//
// register.vin
//
// Simulation of the ARM
// Written by Rónán Dowling-Cullen, Final Year Computer Science 2018/19DLX
// Building on the simulation of the DLX written by Edsko de Vries, Summer 2003

const num HORIZONTAL = 0;
const num VERTICAL = 1;

const num LEFT = 0;
const num RIGHT = 1;
const num TOP = 2;
const num BOTTOM = 3;

Brush regBrush = SolidBrush(PURPLE);
Brush regValueBrush = SolidBrush(WHITE);

class Register(num x, num y, num w, num h, num labelPos, string caption) {

num vx, vy, vw, vh;
num value = 0, newValue = 0;
num tag = 0, newTag = 0;
num useTag = 0, invalid = 0;
num fixed = 0; // for r0
Rectangle label;

Rectangle r1 = Rectangle2(0, 0, blackPen, regBrush, x, y, w, h);
           r1.setRounded(2, 2);
Rectangle bg1 = Rectangle2(valuesLayer, 0, 0, whiteBrush, vx, vy, vw / 2, vh);
Rectangle bg2 = Rectangle2(valuesLayer, 0, 0, whiteBrush, vx + vw / 2, vy, vw / 2, vh);

if (w >= h) {

//
// horizontal
//
vy = y + 4;
vw = w - 38;
vh = h - 9;

if (labelPos == LEFT) {
                                 Rectangle r2 = Rectangle(0, 0, 0, 0, x + 17 - 1, y + h/2, -17, -h/2, 34, h, 0, font, caption);
                                 r2.rotate(-90);
vx = x + 32;
} else if (labelPos == RIGHT) {
r2 = Rectangle(0, 0, 0, 0, x + w - 19, y + h/2, -19, -h/2, 38, h, 0, font, caption);
r2.rotate(-90);
           vx = x + 5;
}

} else {

//
// vertical
//
vx = x + 5;
vw = w - 11;
vh = h - 31;

if (labelPos == TOP) {
r2 = Rectangle2(0, 0, 0, 0, x,y, w,31, 0, font, caption);
vy = y + 26;
} else if (labelPos == BOTTOM) {
r2 = Rectangle2(0, 0, 0, 0, x,y+h-22, w,22, 0, font, caption);
vy = y + 4;
}

}

//
// register value (horizontal or vertical)
//
if (w >= h) {
label = Rectangle2(valuesLayer, 0, 0, yellowBrush, vx, vy, vw, vh, 0, font, "%02X", value);
} else {
label = Rectangle(valuesLayer, 0, 0, yellowBrush, vx + vw/2, vy + vh/2, -vw/2, -vh/2, vw, vh, 0, font, "%02X", value);
}
           label.setRounded(2, 2);

//
// setFixed
//
function setFixed() {
fixed = 1;
}

//
// setOpacity
//
function setOpacity(num opacity) {
r1.setOpacity(opacity);
r2.setOpacity(opacity);
bg1.setOpacity(opacity);
bg2.setOpacity(opacity);
label.setOpacity(opacity);
}

           //
           // rotateText
           //
           function rotateText(num deg){
                      r2.rotate(deg);
           }

//
// updateLabel
//
num function updateLabel() {
if (invalid) {
label.setTxt("INV");
} else if (useTag) {
if (tag >= 0)
label.setTxt("R%d:%02X", tag, value)
else
label.setTxt("--:%02X", value);
} else {
label.setTxt("%02X", value);
}
                      return 0;
}

           //
// mouse enter exit event handler
//
when label ~> eventEE(num enter, num x, num y) {
if (fixed == 0)
label.setBrush(enter ? whiteBrush : yellowBrush);
                      return 0;
}

//
// mouse left button event handler
//
when label ~> eventMB(num down, num flags, num x, num y) {
if (fixed == 0 && down) {
                                 if (flags & MB_LEFT) {
           value = (value + 1) & 0xFF;
                                 } else if (flags & MB_RIGHT) {
                                            value = (value - 1) & 0xFF;
                                 }
updateLabel();
}
                      return 0;
}

//
// setValue
//
function setValue(num val) {
value = val;
invalid = 0;
updateLabel();
}

//
// setNewValue
//
function setNewValue(num val) {
newValue = val;
}

//
// setNewTag
//
function setNewTag(num t) {
newTag = t;
}

//
// setTag
//
function setTag(num t) {
useTag = 1;
tag = t;
updateLabel();
}

//
// update
//
function update() {
value = newValue;
tag = newTag;
updateLabel();
bg1.setBrush(yellowBrush);
bg2.setBrush(yellowBrush);
wait(16);
bg1.setBrush(whiteBrush);
bg2.setBrush(whiteBrush);
}

//
// setInvalid
//
function setInvalid(num i) {
useTag = 1;
invalid = i;
}

//
// tagMatches
//
num function tagMatches(num t) {
return (invalid) ? 0 : (tag == t) ? 1 : 0;
}

num hmode = 0;

//
// highlight
//
function highlight(Brush brush) {
if (hmode == 0) {
bg1.setBrush(brush);
bg2.setBrush(brush);
hmode = 1;
} else {
bg2.setBrush(brush);
}
}

//
// unHighlight
//
function unHighlight() {
bg1.setBrush(whiteBrush);
bg2.setBrush(whiteBrush);
hmode = 0;
}

//
// reset
//
function reset() {
value = 0;
newValue = 0;
tag = 0;
newTag = 0;
useTag = 0;
invalid = 0;
unHighlight();
updateLabel();
}


           //
           // doubleUpdate
           // updates this register and the passedRegister to the same values
           function doubleUpdate(Register reg2) {
                      update();
reg2.value = newValue;
reg2.tag = newTag;
reg2.updateLabel();
reg2.bg1.setBrush(yellowBrush);
reg2.bg2.setBrush(yellowBrush);
reg2.bg1.setBrush(whiteBrush);
reg2.bg2.setBrush(whiteBrush);
           }

}

// eof

//
// schematic.vin
//
// Simulation of the ARM
// Written by Rónán Dowling-Cullen, Final Year Computer Science 2018/19DLX
// Building on the simulation of the DLX written by Edsko de Vries, Summer 2003

string name = getArg("name", "");                                                       // optional name of code example
if (name != "")
           name = sprintf(": %s", name);

Font fTitle = Font("Calibri", 54, SMALLCAPS | ITALIC);

Rectangle title = Rectangle2(0, HLEFT, 0, SolidBrush(DARK_BLUE), 27, 27, 911.0, 66, whitePen, fTitle, sprintf(" ARM9 TDMI Animation
%s", name));

Pen borderPen = SolidPen(DASH, 3, DARK_BLUE, ROUND_START | ROUND_JOIN | ROUND_END);

Line2(0, ABSOLUTE, borderPen, 437.0, 176, 1993.0, 176);
Line2(0, ABSOLUTE, borderPen, 437.0, 1000, 1993.0, 1000);

Line2(0, ABSOLUTE, borderPen, 437.0, 176, 437.0, 1000);
Line b1 = Line2(0, ABSOLUTE, borderPen, 749.0, 176, 749.0, 1000);
Line b2 = Line2(0, ABSOLUTE, borderPen, 1100.0, 176, 1100.0, 1000);
Line b3 = Line2(0, ABSOLUTE, borderPen, 1535.0, 176, 1535.0, 1000);
Line b4 = Line2(0, ABSOLUTE, borderPen, 1825.0, 176, 1825.0, 1000);
Line2(0, ABSOLUTE, borderPen, 1997.0, 176, 1997.0, 1000);

// Buttons

const num BUTTON_PE = 0; // Pipeling enabled
const num BUTTON_BP = 1; // Branch prediction
const num BUTTON_LI = 2; // Load numerlock
const num BUTTON_AF = 3; // ALU forwarding
const num BUTTON_SF = 4; // SMDR forwarding
const num BUTTON_ZF = 5; // Zero forwarding

const num BUTTON_SP = 6; // Safe program

Pen delimeter = SolidPen(DOT, THIN, BLACK);
Line2(0, ABSOLUTE, delimeter, 27, 1008, 2190.0, 1008);

Font arialBold27 = Font("Calibri", 27, BOLD);

Button buttonSP = Button(134, 1014, 216, 54, "Save Configuration", BUTTON_SP);
Button buttonPE = Button(505.0, 1014, 216, 44, "Pipelining Enabled", BUTTON_PE);
Button buttonBP = Button(748.0, 1014, 216, 44, "Branch Prediction", BUTTON_BP);
Button buttonLI = Button(991.0, 1014, 216, 44, "Load Interlock", BUTTON_LI);
Button buttonAF = Button(1235.0, 1014, 216, 44, "ALU Forwarding", BUTTON_AF);
Button buttonSF = Button(1478.0, 1014, 216, 44, "Store Operand\nForwarding", BUTTON_SF);
Button buttonZF = Button(1721.0, 1014, 216, 44, "CPSR Forwarding", BUTTON_ZF);


//
// Vivio logo
//
const num LOGOW = 44;
                                                                  // 64 x 64 image
const num LOGOH = 44;
Image logo = Image(0, 0, 0, "vivio.png", 2146.0,1014, 0,0, LOGOW, LOGOH);

// Statistics

Txt(0, HLEFT | VTOP, 27, 101, darkGrayPen, font, "instructions executed:");
Rectangle et = Txt(0, HLEFT | VTOP, 27, 123, darkGrayPen, font, "ticks:");

Rectangle statIC = Txt(0, HLEFT | VTOP, 243, 101, redPen, font, "0");
Rectangle statTE = Txt(0, HLEFT | VTOP, 243, 123, redPen, font, "0");

// Instruction memory and clock
Rectangle imLabel = Rectangle2(0, 0, 0, 0, 27, 150, 270, 27, 0, font, "Instruction Cache");
InstructionMemory im = InstructionMemory(27, 190, 390, 715);
AnimatedClock animClock = AnimatedClock(54, 914, 216, 66);

// IF and PC
Register regPC = Register(641.0, 463, 54, 88, TOP, "PC");

// branch target buffer
Rectangle btbLabel = Rectangle2(0, 0, 0, 0, 506.0, 187, 216, 22, 0, font, "Branch Target Buffer");
Register btbPC[2];
btbPC[0] = Register(506.0, 220, 108, 44, LEFT, "PC");
btbPC[1] = Register(506.0, 264, 108, 44, LEFT, "PC");
Register btbPPC[2];
btbPPC[0] = Register(614.0, 220, 108, 44, RIGHT, "PPC");
btbPPC[1] = Register(614.0, 264, 108, 44, RIGHT, "PPC");

// Other components
Component cMUX2 = Component(641.0, 375, 81, 22, "mux 2");
Component cMUX1 = Component(560.0, 452, 27, 110, "mux 1");
Component cPLUS4 = Component(533.0, 595, 54, 22, "+4");

// connections from and to instruction memory
```

```
AnimPipe apIM_ID = AnimPipe();
apIM_ID.addPoint(415.0, 860);
apIM_ID.addPoint(776.0, 860);
Rectangle txtIM_ID = Rectangle(activePipesLayer, 0, 0, redBrush, 587.0,8130, -81,-13, 130,26, whitePen, font);
txtIM_ID.setRounded(2, 2);

AnimPipe apPC_IM = AnimPipe();
apPC_IM.addPoint(668.0, 551);
apPC_IM.addPoint(668.0, 705);
apPC_IM.addPoint(415.0, 705);

// connections into MUX1
AnimPipe apMUX3_MUX1 = AnimPipe();
apMUX3_MUX1.addPoint(911.0, 375);
apMUX3_MUX1.addPoint(911.0, 353);
apMUX3_MUX1.addPoint(506.0, 353);
apMUX3_MUX1.addPoint(506.0, 474);
apMUX3_MUX1.addPoint(560.0, 474);

AnimPipe apBTB_MUX1 = AnimPipe();
apBTB_MUX1.addPoint(506.0, 264);
apBTB_MUX1.addPoint(479.0, 264);
apBTB_MUX1.addPoint(479.0, 496);
apBTB_MUX1.addPoint(560.0, 496);

AnimPipe apRF_MUX1 = AnimPipe();
apRF_MUX1.addPoint(952.0, 110);
apRF_MUX1.addPoint(452.0, 110);
apRF_MUX1.addPoint(452.0, 518);
apRF_MUX1.addPoint(560.0, 518);

AnimPipe apADD4_MUX1 = AnimPipe();
apADD4_MUX1.addPoint(533.0, 606);
apADD4_MUX1.addPoint(506.0, 606);
apADD4_MUX1.addPoint(506.0, 540);
apADD4_MUX1.addPoint(560.0, 540);

// Connections to and from PC
AnimPipe apMUX1_PC = AnimPipe();
apMUX1_PC.addPoint(587.0, 507);
apMUX1_PC.addPoint(641.0, 507);

AnimPipe apPC_MUX2 = AnimPipe();
apPC_MUX2.addPoint(668.0, 462);
apPC_MUX2.addPoint(668.0, 397);

AnimPipe apPC_ADD4 = AnimPipe();
apPC_ADD4.addPoint(668.0, 551);
apPC_ADD4.addPoint(668.0, 606);
apPC_ADD4.addPoint(587.0, 606);

AnimPipe apPC_PC1 = AnimPipe();
apPC_PC1.addPoint(695.0, 507);
apPC_PC1.addPoint(749.0, 507);
apPC_PC1.addPoint(776.0, 507);

// MUX2 to BTB
AnimPipe apMUX2_BTB = AnimPipe();
apMUX2_BTB.addPoint(681.0, 375);
apMUX2_BTB.addPoint(681.0, 309);

//
// Instruction Decode
//
// ID to EX
//
AnimPipe apID_EX = AnimPipe();
apID_EX.addPoint(830.0, 860);
apID_EX.addPoint(1132.0, 860);

// ID and PC1
InstructionRegister regID = InstructionRegister(776.0, 745, 54, 250, "ID");
Register regPC1 = Register(776.0, 463, 54, 88, TOP, "PC1");

// Register file
Txt(0, HLEFT | VTOP, 1182.0, 88, 0, font, "Register\nFile");
Register regFile[16];
regFile[0] = Register(952.0, 66, 32, 88, TOP, "R0");
regFile[1] = Register(984.0, 66, 32, 88, TOP, "R1");
regFile[2] = Register(1016.0, 66, 32, 88, TOP, "R2");
regFile[3] = Register(1048.0, 66, 32, 88, TOP, "R3");
regFile[4] = Register(1080.0, 66, 32, 88, TOP, "R4");
regFile[5] = Register(1112.0, 66, 32, 88, TOP, "R5");
```

```
regFile[6] = Register(1144.0, 66, 32, 88, TOP, "R6");
regFile[7] = Register(1176.0, 66, 32, 88, TOP, "R7");
regFile[8] = Register(1208.0, 66, 32, 88, TOP, "R8");
regFile[9] = Register(1240.0, 66, 32, 88, TOP, "R9");
regFile[10] = Register(1272.0, 66, 32, 88, TOP, "R10");
regFile[11] = Register(1304.0, 66, 32, 88, TOP, "R11");
regFile[12] = Register(1336.0, 66, 32, 88, TOP, "R12");
regFile[13] = Register(1368.0, 66, 32, 88, TOP, "SP");
regFile[14] = Register(1400.0, 66, 32, 88, TOP, "LR");
regFile[15] = Register(1432.0, 66, 32, 88, TOP, "PC");

//Initialise SP and PC
regFile[13].setValue(0x40);
regFile[15].setValue(0x7C);

//
// CPSR
//

CPSR cpsr = CPSR(1490, 66, 140, 88, 0);

// Other components
Component cMUX3 = Component(844.0, 375, 135, 22, "mux 3");
Component cADD4 = Component(830.0, 595, 81, 22, "ADD4");
Component cADDi = Component(911.0, 595, 81, 22, "ADDi");
Component cMUX4 = Component(776.0, 220, 27, 88, "mux 4");
Component cMUX5 = Component(1051.0, 518, 27, 66, "mux 5");

Rectangle2(0, 0, 0, 0, 857.0, 661, 54, 22, 0, font, "4");

// Control lines for MUX3
Pen controlPen = SolidPen(SOLID, 2, PURPLE, ARROW60_END);
Pen controlHLPen = SolidPen(SOLID, 5, RED, ARROW60_END);
Pen controlMuxPen = SolidPen(SOLID, 8, GRAY192);
Pen controlMuxHLPen = SolidPen(SOLID, 8, RED);
Pen muxPen = SolidPen(SOLID, THICK, BLACK);

// Connections from PC1
AnimPipe apPC1_MUX2 = AnimPipe();
apPC1_MUX2.addPoint(803.0, 463);
apPC1_MUX2.addPoint(803.0, 441);
apPC1_MUX2.addPoint(695.0, 441);
apPC1_MUX2.addPoint(695.0, 397);

AnimPipe apPC1_MUX5 = AnimPipe();
apPC1_MUX5.addPoint(803.0, 551);
apPC1_MUX5.addPoint(803.0, 565);
apPC1_MUX5.addPoint(1051.0, 565);

AnimPipe apPC1_ADD4 = AnimPipe();
apPC1_ADD4.addPoint(803.0, 551);
apPC1_ADD4.addPoint(803.0, 705);
apPC1_ADD4.addPoint(857.0, 705);
apPC1_ADD4.addPoint(857.0, 617);

AnimPipe apPC1_ADDi = AnimPipe();
apPC1_ADDi.addPoint(803.0, 551);
apPC1_ADDi.addPoint(803.0, 705);
apPC1_ADDi.addPoint(938.0, 705);
apPC1_ADDi.addPoint(938.0, 617);

// Connections to and from the adders
AnimPipe ap4_ADD4 = AnimPipe();
ap4_ADD4.addPoint(884.0, 661);
ap4_ADD4.addPoint(884.0, 617);

AnimPipe apID_ADDi = AnimPipe();
apID_ADDi.addPoint(830.0, 860);
apID_ADDi.addPoint(965.0, 860);
apID_ADDi.addPoint(965.0, 617);
Rectangle txtID_ADDi = Rectangle(activePipesLayer, 0, 0, redBrush, 8235.0,829, -32,-13, 65,32, whitePen, font);
txtID_ADDi.setRounded(2, 2);

AnimPipe apADD4_MUX3 = AnimPipe();
apADD4_MUX3.addPoint(871.0, 595);
apADD4_MUX3.addPoint(871.0, 397);
Rectangle txtADD4_MUX3 = Rectangle(activePipesLayer, 0, 0, redBrush, 871.0,441, -32,-13, 65,32, whitePen, font);
txtADD4_MUX3.setRounded(2, 2);

AnimPipe apADDi_MUX3 = AnimPipe();
apADDi_MUX3.addPoint(952.0, 595);
apADDi_MUX3.addPoint(952.0, 397);
Rectangle txtADDi_MUX3 = Rectangle(activePipesLayer, 0, 0, redBrush, 952.0,441, -32,-13, 65,32, whitePen, font);
```

```
txtADDi_MUX3.setRounded(2, 2);

// Connection to BTB
AnimPipe apMUX3_MUX4 = AnimPipe();
apMUX3_MUX4.addPoint(911.0, 375);
apMUX3_MUX4.addPoint(911.0, 287);
apMUX3_MUX4.addPoint(803.0, 287);
Rectangle txtMUX3_MUX4 = Rectangle(activePipesLayer, 0, 0, redBrush, 860.0,320, -32,-13, 65,32, whitePen, font);
txtMUX3_MUX4.setRounded(2, 2);

AnimPipe apMUX4_BTB = AnimPipe();
apMUX4_BTB.addPoint(776.0, 264);
apMUX4_BTB.addPoint(722.0, 264);

AnimPipe apRF_MUX4 = AnimPipe();
apRF_MUX4.addPoint(952.0, 110);
apRF_MUX4.addPoint(911.0, 110);
apRF_MUX4.addPoint(911.0, 242);
apRF_MUX4.addPoint(803.0, 242);
Rectangle txtRF_MUX1 = Rectangle(activePipesLayer, 0, 0, redBrush, 911.0,97, -32,0, 65,32, whitePen, font);

// MUX5 to A
AnimPipe apMUX5_B = AnimPipe();
apMUX5_B.addPoint(1078.0, 551);
apMUX5_B.addPoint(1132.0, 551);

// Connections from the register file
AnimPipe apRF_MUX5 = AnimPipe();
apRF_MUX5.addPoint(1000.0, 154);
apRF_MUX5.addPoint(1000.0, 540);
apRF_MUX5.addPoint(1051.0, 540);
Rectangle txtRF_MUX5 = Rectangle(activePipesLayer, 0, 0, redBrush, 990.0,181, -32,0, 65,32, whitePen, font, "R0:0");
txtRF_MUX5.setRounded(2, 2);

AnimPipe apRF_A = AnimPipe();
apRF_A.addPoint(1051.0, 154);
apRF_A.addPoint(1051.0, 463);
apRF_A.addPoint(1082.0, 463);
Rectangle txtRF_A = Rectangle(activePipesLayer, 0, 0, redBrush, 1065.0,181, -32,0, 65,32, whitePen, font, "R0:0");
txtRF_A.setRounded(2, 2);

//
// Execution
//
// EX, A and B
//
InstructionRegister regEX = InstructionRegister(1132.0, 745, 54, 250, "EX");
Register regA = Register(1132.0, 419, 54, 88, TOP, "A");
Register regB = Register(1132.0, 507, 54, 88, BOTTOM, "B");

// Other components
Component cMUX6 = Component(1301.0, 397, 27, 110, "mux 6");
Component cMUX7 = Component(1301.0, 507, 27, 110, "mux 7");
Component cMUX8 = Component(1301.0, 675, 27, 74, "mux 8");
ALU alu = ALU(1445.0, 419, 69, 176);

// EX to MA
AnimPipe apEX_MA = AnimPipe();
apEX_MA.addPoint(1186.0, 860);
apEX_MA.addPoint(1614.0, 860);

// Connections numo MUX6
AnimPipe apOUT0_MUX6 = AnimPipe();
apOUT0_MUX6.addPoint(1614.0, 480);
apOUT0_MUX6.addPoint(1614.0, 375);
apOUT0_MUX6.addPoint(1267.0, 375);
apOUT0_MUX6.addPoint(1267.0, 419);
apOUT0_MUX6.addPoint(1301.0, 419);

AnimPipe apOUT1_MUX6 = AnimPipe();
apOUT1_MUX6.addPoint(1896.0, 480);
apOUT1_MUX6.addPoint(1896.0, 353);
apOUT1_MUX6.addPoint(1240.0, 353);
apOUT1_MUX6.addPoint(1240.0, 441);
apOUT1_MUX6.addPoint(1301.0, 441);

AnimPipe apA_MUX6 = AnimPipe();
apA_MUX6.addPoint(1186.0, 485);
apA_MUX6.addPoint(1301.0, 485);

// Connections numo MUX7
AnimPipe apB_MUX7 = AnimPipe();
apB_MUX7.addPoint(1186.0, 529);
apB_MUX7.addPoint(1301.0, 529);

AnimPipe apEX_MUX7 = AnimPipe();
apEX_MUX7.addPoint(1186.0, 860);
apEX_MUX7.addPoint(1213.0, 860);
apEX_MUX7.addPoint(1213.0, 551);
apEX_MUX7.addPoint(1371.0, 551);
Rectangle txtEX_MUX7 = Rectangle(activePipesLayer, 0, 0, redBrush, 1268.0,816, -27,0, 54,26, whitePen, font, "IMM");
txtEX_MUX7.setRounded(2, 2);

//Animations to Barrel Shifter
AnimPipe apEX_BS = AnimPipe();
apEX_BS.addPoint(1186.0, 860);
apEX_BS.addPoint(1388.0, 860);
apEX_BS.addPoint(1388.0, 607);
Rectangle txtEX_BS = Rectangle(activePipesLayer, 0, 0, redBrush, 1388.0,816, -27,0, 54,26, whitePen, font, "IMM");
txtEX_BS.setRounded(2, 2);

AnimPipe apRF_BS = AnimPipe();
apRF_BS.addPoint(1388.0, 154);
apRF_BS.addPoint(1388.0, 517);
Rectangle txtRF_BS = Rectangle(activePipesLayer, 0, 0, redBrush, 1388.0,181, -32,0, 65,32, whitePen, font, "R0:0");
txtRF_BS.setRounded(2, 2);

AnimPipe apOUT1_MUX7 = AnimPipe();
apOUT1_MUX7.addPoint(1896.0, 534);
apOUT1_MUX7.addPoint(1896.0, 661);
apOUT1_MUX7.addPoint(1240.0, 661);
apOUT1_MUX7.addPoint(1240.0, 573);
apOUT1_MUX7.addPoint(1301.0, 573);

AnimPipe apOUT0_MUX7 = AnimPipe();
apOUT0_MUX7.addPoint(1614.0, 534);
apOUT0_MUX7.addPoint(1614.0, 639);
apOUT0_MUX7.addPoint(1267.0, 639);
apOUT0_MUX7.addPoint(1267.0, 595);
apOUT0_MUX7.addPoint(1301.0, 595);

// Connectings to and from MUX8
AnimPipe apOUT0_MUX8 = AnimPipe();
apOUT0_MUX8.addPoint(1614.0, 534);
apOUT0_MUX8.addPoint(1614.0, 639);
apOUT0_MUX8.addPoint(1267.0, 639);
apOUT0_MUX8.addPoint(1267.0, 690);
apOUT0_MUX8.addPoint(1301.0, 690);

AnimPipe apOUT1_MUX8 = AnimPipe();
apOUT1_MUX8.addPoint(1896.0, 534);
apOUT1_MUX8.addPoint(1896.0, 661);
apOUT1_MUX8.addPoint(1240.0, 661);
apOUT1_MUX8.addPoint(1240.0, 712);
apOUT1_MUX8.addPoint(1301.0, 712);

AnimPipe apB_MUX8 = AnimPipe();
apB_MUX8.addPoint(1159.0, 595);
apB_MUX8.addPoint(1159.0, 734);
apB_MUX8.addPoint(1301.0, 734);

AnimPipe apMUX8_SMDR = AnimPipe();
apMUX8_SMDR.addPoint(1328.0, 712);
apMUX8_SMDR.addPoint(1657.0, 712);


//
// Connections from MUX6 through the ALU to OUT0
//
AnimPipe apMUX6_OUT0 = AnimPipe();
apMUX6_OUT0.addPoint(1328.0, 452);
apMUX6_OUT0.addPoint(1445.0, 452);
apMUX6_OUT0.addPoint(1530.0, 507);
apMUX6_OUT0.addPoint(1570.0, 507);

AnimPipe apMUX7_OUT0 = AnimPipe();
apMUX7_OUT0.addPoint(1328.0, 562);
apMUX7_OUT0.addPoint(1445.0, 562);
apMUX7_OUT0.addPoint(1530.0, 507);
apMUX7_OUT0.addPoint(1570.0, 507);

//Barrel Shifter
Component BarrelShifter = Component(1361.0, 517, 55, 90, "Barrel\nShifter");


//
```

```
// Memory access
//
// MA, SMDR and OUT0
//

// CPSR Forwarding
CPSR cpsr0 = CPSR(1660.0, 477, 40, 60, 1);
CPSR cpsr1 = CPSR(1938.0, 477, 40, 60, 1);


// CPSR Forwarding Connections
Line cpsrOUT0 = Line2(activePipesLayer, ABSOLUTE, controlPen, 1680.0,477, 1680.0,450, 1545.0,450, 1545.0,480);
Line cpsrOUT1 = Line2(activePipesLayer, ABSOLUTE, controlPen, 1958.0,477, 1958.0,420, 1545.0,420, 1545.0,480);
Line cpsrCPSR = Line2(activePipesLayer, ABSOLUTE, controlPen, 1545.0,154, 1545.0,480);
Line cpsrEX = Line2(activePipesLayer, ABSOLUTE, controlPen, 1186,820, 1545.0,820, 1545.0,480);

Line conditionMux = Line2(activePipesLayer, ABSOLUTE, controlPen, 1545.0, 480, 1545.0, 533);
Rectangle conditionMuxTxt = Txt(valuesLayer,HLEFT | VTOP, 1550.0, 620, 0, sFont, "Condition\nCheck");
conditionMuxTxt.rotate(-90);

Line cpsrOUT0B = Line2(activePipesLayer, ABSOLUTE, controlPen, 1680.0,477, 1680.0,300, 929.0,300, 929.0,365);
Line cpsrOUT1B = Line2(activePipesLayer, ABSOLUTE, controlPen, 1958.0,477, 1958.0,310, 935.0,310, 935,365);
Line cpsrCPSRB = Line2(activePipesLayer, ABSOLUTE, controlPen, 1545.0,154, 1545.0,290, 923.0,290, 923.0,365);
Line cpsrEXB = Line2(activePipesLayer, ABSOLUTE, controlPen, 1132,820, 1110.0,820, 1110.0,365, 935,365);

Line conditionMuxB = Line2(activePipesLayer, ABSOLUTE, controlPen, 875.0, 365, 945.0, 365);
Rectangle conditionMuxBTxt = Txt(valuesLayer,HLEFT | VTOP, 1550.0, 620, 0, sFont, "Condition\nCheck");
conditionMuxBTxt.rotate(-90);

InstructionRegister regMA = InstructionRegister(1614.0, 745, 54, 250, "MA");
Register regOUT0 = Register(1570.0, 480, 88, 54, LEFT, "O0");
Register regSMDR = Register(1587.0, 690, 108, 44, RIGHT, "SMR");
Rectangle memAddrTxt = Txt(valuesLayer, HLEFT | VTOP, 1660.0, 290, 0, sFont, "memory\naddress");
memAddrTxt.rotate(-90);
Txt(valuesLayer, HLEFT | VTOP, 1750.0, 705, 0, sFont, "memory\ndata-in");
Rectangle memDataOut = Txt(valuesLayer, HLEFT | VTOP, 1810.0, 220, 0, sFont, "memory\ndata-out");
memDataOut.rotate(90);

// Memory
Txt(0, HLEFT | VTOP, 1695.0, 32, 0, sFont, "Data Cache (memory)");
Register memory[16];
memory[0] = Register(1654.0, 58, 70, 25, LEFT, "00");
memory[1] = Register(1654.0, 83, 70, 25, LEFT, "04");
memory[2] = Register(1654.0, 108, 70, 25, LEFT, "08");
memory[3] = Register(1654.0, 133, 70, 25, LEFT, "0C");
memory[4] = Register(1724.0, 58, 70, 25, LEFT, "10");
memory[5] = Register(1724.0, 83, 70, 25, LEFT, "14");
memory[6] = Register(1724.0, 108, 70, 25, LEFT, "18");
memory[7] = Register(1724.0, 133, 70, 25, LEFT, "1C");

//Stack Memory
memory[8] = Register(1794.0, 58, 70, 25, LEFT, "20");
memory[9] = Register(1794.0, 83, 70, 25, LEFT, "24");
memory[10] = Register(1794.0, 108, 70, 25, LEFT, "28");
memory[11] = Register(1794.0, 133, 70, 25, LEFT, "2C");
memory[12] = Register(1864.0, 58, 70, 25, LEFT, "30");
memory[13] = Register(1864.0, 83, 70, 25, LEFT, "34");
memory[14] = Register(1864.0, 108, 70, 25, LEFT, "38");
memory[15] = Register(1864.0, 133, 70, 25, LEFT, "3C");

memory[0].rotateText(90);
memory[1].rotateText(90);
memory[2].rotateText(90);
memory[3].rotateText(90);
memory[4].rotateText(90);
memory[5].rotateText(90);
memory[6].rotateText(90);
memory[7].rotateText(90);
memory[8].rotateText(90);
memory[9].rotateText(90);
memory[10].rotateText(90);
memory[11].rotateText(90);
memory[12].rotateText(90);
memory[13].rotateText(90);
memory[14].rotateText(90);
memory[15].rotateText(90);


//Stack
Rectangle outer = Rectangle2(0, 0, blackPen, 0, 2010, 500, 180, 500);
outer.setRounded(2, 2);

Txt(0, HLEFT | VTOP, 2015.0, 517, 0, font, "Stack Visualisation");
```

```
Register stack[8];
stack[7] = Register(2055.0, 585, 100, 50, LEFT, "3C");
stack[6] = Register(2055.0, 635, 100, 50, LEFT, "38");
stack[5] = Register(2055.0, 685, 100, 50, LEFT, "34");
stack[4] = Register(2055.0, 735, 100, 50, LEFT, "30");
stack[3] = Register(2055.0, 785, 100, 50, LEFT, "2C");
stack[2] = Register(2055.0, 835, 100, 50, LEFT, "28");
stack[1] = Register(2055.0, 885, 100, 50, LEFT, "24");
stack[0] = Register(2055.0, 935, 100, 50, LEFT, "20");

stack[0].rotateText(90);
stack[1].rotateText(90);
stack[2].rotateText(90);
stack[3].rotateText(90);
stack[4].rotateText(90);
stack[5].rotateText(90);
stack[6].rotateText(90);
stack[7].rotateText(90);

//Stack Pointer
StackPointer sp = StackPointer(2020,615,2050,615,12,50);

//
// Other components
//
Component cMUX9 = Component(1782.0, 463, 27, 88, "mux 9");

// MA to WB
AnimPipe apMA_WB = AnimPipe();
apMA_WB.addPoint(1668.0, 860);
apMA_WB.addPoint(1885.0, 860);

// OUT0 to MUX 8/OUT1
//
AnimPipe apOUT0_MUX9 = AnimPipe();
apOUT0_MUX9.addPoint(1700.0, 507);
apOUT0_MUX9.addPoint(1782.0, 507);

AnimPipe apMUX9_OUT1 = AnimPipe();
apMUX9_OUT1.addPoint(1809.0, 507);
apMUX9_OUT1.addPoint(1856.0, 507);

// Connections to and from memory

AnimPipe apOUT0_Mem = AnimPipe();
apOUT0_Mem.addPoint(1700.0, 507);
apOUT0_Mem.addPoint(1715.0, 507);
apOUT0_Mem.addPoint(1715.0, 158);

AnimPipe apSMDR_Mem = AnimPipe();
apSMDR_Mem.addPoint(1695.0, 727);
apSMDR_Mem.addPoint(1737.0, 727);
apSMDR_Mem.addPoint(1737.0, 158);

AnimPipe apMem_MUX9 = AnimPipe();
apMem_MUX9.addPoint(1760.0, 158);
apMem_MUX9.addPoint(1760.0, 485);
apMem_MUX9.addPoint(1782.0, 485);

//
// Write Back
//

// WB and OUT1

InstructionRegister regWB = InstructionRegister(1885.0, 745, 54, 250, "WB");
Register regOUT1 = Register(1848.0, 480, 88, 54, LEFT, "O1");



// Connections from OUT1 to the register file

AnimPipe apOUT1_RF = AnimPipe();
apOUT1_RF.addPoint(1978.0, 507);
apOUT1_RF.addPoint(1987.0, 507);
apOUT1_RF.addPoint(1987.0, 22);
apOUT1_RF.addPoint(1207.0, 22);
apOUT1_RF.addPoint(1207.0, 66);

AnimPipe apCPSR1_RF = AnimPipe();
apCPSR1_RF.addPoint(1978.0, 507);
apCPSR1_RF.addPoint(1987.0, 507);
apCPSR1_RF.addPoint(1987.0, 22);
```

```
apCPSR1_RF.addPoint(1560.0, 22);
apCPSR1_RF.addPoint(1560.0, 66);

alu.txtResult.moveToFront();

//
// ResetWires
//
function resetWires() {

          //
          // IF
          //
apPC_IM.reset();
apIM_ID.reset();
txtIM_ID.setOpacity(0);
apMUX3_MUX1.reset();
apBTB_MUX1.reset();
apRF_MUX1.reset();
apADD4_MUX1.reset();
apMUX1_PC.reset();
apPC_MUX2.reset();
apPC_ADD4.reset();
apPC_PC1.reset();
apMUX2_BTB.reset();

//
          // ID
          //
apID_EX.reset();
apPC1_MUX2.reset();
          apPC1_MUX5.reset()
apPC1_ADD4.reset();
apPC1_ADDi.reset();
ap4_ADD4.reset();
apID_ADDi.reset(); txtID_ADDi.setOpacity(0);
apADD4_MUX3.reset(); txtADD4_MUX3.setOpacity(0);
apADDi_MUX3.reset(); txtADDi_MUX3.setOpacity(0);
apMUX3_MUX4.reset(); txtMUX3_MUX4.setOpacity(0);
apRF_MUX4.reset();
apMUX4_BTB.reset();
apMUX5_B.reset();
apRF_MUX5.reset(); txtRF_MUX5.setOpacity(0);
apRF_A.reset(); txtRF_A.setOpacity(0);
txtRF_MUX1.setOpacity(0);
//
// EX
//
apEX_MA.reset();
apOUT0_MUX6.reset();
apOUT1_MUX6.reset();
apA_MUX6.reset();
apB_MUX7.reset();
apEX_MUX7.reset(); txtEX_MUX7.setOpacity(0);
          apEX_BS.reset(); txtEX_BS.setOpacity(0);
apRF_BS.reset(); txtRF_BS.setOpacity(0);
apOUT1_MUX7.reset();
apOUT0_MUX7.reset();
apOUT0_MUX8.reset();
apOUT1_MUX8.reset();
apB_MUX8.reset();
apMUX8_SMDR.reset();
apMUX6_OUT0.reset();
apMUX7_OUT0.reset();
alu.txtOp.setOpacity(0);
alu.txtResult.setOpacity(0);

//
// MA
//
apMA_WB.reset();
apOUT0_MUX9.reset();
apMUX9_OUT1.reset();
apOUT0_Mem.reset();
apSMDR_Mem.reset();
apMem_MUX9.reset();
          cpsrOUT0.setPen(controlPen);
          cpsrOUT1.setPen(controlPen);
          cpsrCPSR.setPen(controlPen);
          cpsrEX.setPen(controlPen);
          cpsrOUT0B.setPen(controlPen);
          cpsrOUT1B.setPen(controlPen);
          cpsrCPSRB.setPen(controlPen);
          cpsrEXB.setPen(controlPen);
          conditionMux.setPen(controlMuxPen);
          conditionMuxB.setPen(controlMuxPen);

//
// WB
//
apOUT1_RF.reset();
          apCPSR1_RF.reset();

}

//
// resetRegisters()
//
function resetRegisters() {

regPC.reset(); regPC.setValue(0x7C);
regPC1.reset();
regA.reset();
regB.reset();
regSMDR.reset();
regOUT0.reset();
regOUT1.reset();

          cpsr0.reset();
          cpsr1.reset();

btbPC[0].reset();
btbPC[1].reset();
btbPPC[0].reset();
btbPPC[1].reset();


regID.reset();
regEX.reset();
regMA.reset();
regWB.reset();

im.setActive(0x7C);

regOUT0.setInvalid(1);
regOUT0.updateLabel();
regOUT1.setInvalid(1);
regOUT1.updateLabel();
btbPC[0].setValue(-1);
btbPC[0].setInvalid(1);
btbPC[0].updateLabel();
btbPC[1].setValue(-1);
btbPC[1].setInvalid(1);
btbPC[1].updateLabel();
          cpsr0.setInvalid(1);
          cpsr1.setInvalid(1);
          instrCount = 0;
tickCount = 0;

statIC.setTxt("%4d", 0);
statTE.setTxt("%4d", 0);

}

function resetCircuit() {
resetRegisters();
resetWires();
}

//
// Enabling and disabling parts of the pipeline
//
function showBTB(num opacity) {

btbLabel.setOpacity(opacity);
btbPC[0].setOpacity(opacity);
btbPC[1].setOpacity(opacity);
btbPPC[0].setOpacity(opacity);
btbPPC[1].setOpacity(opacity);

apPC_MUX2.setOpacity(opacity);
apPC1_MUX2.setOpacity(opacity);
cMUX2.setOpacity(opacity);
apMUX2_BTB.setOpacity(opacity);

apBTB_MUX1.setOpacity(opacity);
```

```
apMUX3_MUX4.setOpacity(opacity);
apRF_MUX4.setOpacity(opacity);
cMUX4.setOpacity(opacity);
apMUX4_BTB.setOpacity(opacity);

}

//
// showALUForwarding
//
function showALUForwarding(num opacity) {

if (opacity == 0) {

//
// NOT pipelined
//
apA_MUX6.setPoint(0, 1186.0, 452);
apA_MUX6.setPoint(1, 1351.0, 452);

apB_MUX7.setPoint(0, (afMode) ? 1186 : 1209.0, 551);
apB_MUX7.setPoint(1, 1301.0, 551);

apEX_MUX7.setPoint(2, 1213.0, 573);
apEX_MUX7.setPoint(3, 1301.0, 573);

apA_MUX6.setHead(0);

} else {

apA_MUX6.setPoint(0, 1186.0, 485);
apA_MUX6.setPoint(1, 1301.0, 485);

apB_MUX7.setPoint(0, 1186.0, 529);
apB_MUX7.setPoint(1, 1301.0, 529);

apEX_MUX7.setPoint(2, 1213.0, 551);
apEX_MUX7.setPoint(3, 1301.0, 551);

apA_MUX6.setHead(1);

}

cMUX6.setOpacity(opacity);
apOUT0_MUX6.setOpacity(opacity);
apOUT1_MUX6.setOpacity(opacity);

apOUT0_MUX7.setOpacity(opacity);
apOUT1_MUX7.setOpacity(opacity);

}

//
// showSMDRForwarding
//
function showSMDRForwarding(num opacity) {

if (opacity == 0) {

apB_MUX8.setPoint(1, 1159.0, 712);
apB_MUX8.setPoint(2, 1328.0, 712);
apB_MUX8.setHead(0);

} else {

apB_MUX8.setPoint(1, 1159.0, 734);
apB_MUX8.setPoint(2, 1301.0, 734);
apB_MUX8.setHead(1);

}

cMUX8.setOpacity(opacity);
apOUT0_MUX8.setOpacity(opacity);
apOUT1_MUX8.setOpacity(opacity);

}



//
// showCPSRForwarding
//
function showCPSRForwarding(num opacity) {

        // CPSR Forwarding
        cpsr0.setOpacity(opacity);
        cpsr1.setOpacity(opacity);


        // CPSR Forwarding Connections
        cpsrOUT0.setOpacity(opacity);
        cpsrOUT1.setOpacity(opacity);
        cpsrOUT0B.setOpacity(opacity);
        cpsrOUT1B.setOpacity(opacity);

        if(opacity == 0){
                cpsrEX.setPt(0,-356,0);
                cpsrEXB.setPt(0,-306,0);
        } else {
                cpsrEX.setPt(0,0,0);
                cpsrEXB.setPt(0,0,0);
        }

}


//
// showPipeline
//
function showPipeline(num opacity) {

if (opacity == 0) {

//
// not pipelined
//
apPC_PC1.setPoint(1, 803.0, 507);
apPC_PC1.setPoint(2, 803.0, 529);

apPC1_ADD4.setPoint(0, 803.0, 507);
apPC1_ADDi.setPoint(0, 803.0, 507);

apID_EX.setPoint(1, 1186.0, 860);

apRF_A.setPoint(1, 1051.0, 452);
apRF_A.setPoint(2, 1186.0, 452);
apMUX5_B.setPoint(1, 1209.0, 551);
apB_MUX8.setPoint(0, 1159.0, 551);

apMUX6_OUT0.setPoint(3, 1570.0, 507);
apMUX7_OUT0.setPoint(3, 1570.0, 507);
apMUX8_SMDR.setPoint(0, 1348.0, 727);
apMUX8_SMDR.setPoint(1, 1695.0, 727);
apMUX9_OUT1.setPoint(1, 1939.0, 507);

                apOUT0_MUX9.setPoint(0, 1570.0, 507);
                apOUT1_RF.setPoint(0, 1848.0, 507);

                apEX_BS.setPoint(0,830.0, 860);
                apEX_BS.setPoint(1,1388.0, 860);

apID_EX.setHead(0);
apPC_PC1.setHead(0);
apRF_A.setHead(0);
apA_MUX6.setHead(0);
apMUX5_B.setHead(0);
apB_MUX8.setHead(0);
apMUX8_SMDR.setHead(0);
apMUX6_OUT0.setHead(0);
apMUX7_OUT0.setHead(0);
apMUX9_OUT1.setHead(0);

showBTB(opacity);
showALUForwarding(opacity);
showSMDRForwarding(opacity);
showCPSRForwarding(opacity);

} else {

apPC_PC1.setPoint(1, 749.0, 507);
apPC_PC1.setPoint(2, 776.0, 507);

apPC1_ADD4.setPoint(0, 803.0, 551);
apPC1_ADDi.setPoint(0, 803.0, 551);
apID_EX.setPoint(1, 1132.0, 860);

apRF_A.setPoint(1, 1051.0, 463);
apRF_A.setPoint(2, 1132.0, 463);
apMUX5_B.setPoint(1, 1132.0, 551);
apB_MUX8.setPoint(0, 1159.0, 595);

apMUX6_OUT0.setPoint(3, 1570.0, 507);
apMUX7_OUT0.setPoint(3, 1570.0, 507);
apMUX8_SMDR.setPoint(0, 1328.0, 712);
apMUX8_SMDR.setPoint(1, 1587.0, 712);
apMUX9_OUT1.setPoint(1, 1848.0, 507);

                      apOUT0_MUX9.setPoint(0, 1700.0, 507);
                      apOUT1_RF.setPoint(0, 1978.0, 507);

                      apEX_BS.setPoint(0,1186.0, 860);
                      apEX_BS.setPoint(1,1388.0, 860);

apID_EX.setHead(1);
apPC_PC1.setHead(1);
apRF_A.setHead(1);
apA_MUX6.setHead(1);
apMUX5_B.setHead(1);
apB_MUX8.setHead(1);
apMUX8_SMDR.setHead(1);
apMUX6_OUT0.setHead(1);
apMUX7_OUT0.setHead(1);
apMUX9_OUT1.setHead(1);

showBTB(bpMode == BRANCH_PREDICTION ? 1 : 0);
showALUForwarding(afMode == ALU_FORWARDING ? 1 : 0);
showSMDRForwarding(sfMode == FORWARDING_TO_SMDR ? 1 : 0);
showCPSRForwarding(zfMode == ZERO_FORWARDING ? 1 : 0);

}

apPC_ADD4.setOpacity(opacity);
cPLUS4.setOpacity(opacity);
apADD4_MUX1.setOpacity(opacity);

regPC1.setOpacity(opacity);
regEX.setOpacity(opacity);
regMA.setOpacity(opacity);
regWB.setOpacity(opacity);

apEX_MA.setOpacity(opacity);
apMA_WB.setOpacity(opacity);

regA.setOpacity(opacity);
regB.setOpacity(opacity);
regOUT0.setOpacity(opacity);
regOUT1.setOpacity(opacity);
regSMDR.setOpacity(opacity);

buttonBP.label.setOpacity(opacity);
buttonLI.label.setOpacity(opacity);
buttonAF.label.setOpacity(opacity);
buttonSF.label.setOpacity(opacity);
buttonZF.label.setOpacity(opacity);

}

//
// setPEMode
//
function setPEMode(num mode) {
peMode = mode;
if (peMode == 0) {
buttonPE.setCaption("Pipelining Enabled");
showPipeline(1);
} else if (peMode == 1) {
buttonPE.setCaption("Pipelining Disabled");
showPipeline(0);
}
setArg("peMode", peMode.toString());
}

//
// setBPMode
//
function setBPMode(num mode) {
bpMode = mode;
if (bpMode == 0) {
buttonBP.setCaption("Branch Prediction");
showBTB(1);
} else if (bpMode == 1) {
buttonBP.setCaption("Branch Interlock");
showBTB(0);
} else if (bpMode == 2) {
buttonBP.setCaption("Delayed Branches");
showBTB(0);
}
setArg("bpMode", bpMode.toString());
}

//
// setLIMode
//
function setLIMode(num mode) {
liMode = mode;
if (liMode == 0) {
buttonLI.setCaption("Load Interlock");
} else if (liMode == 1) {
buttonLI.setCaption("No Load Interlock");
}
setArg("liMode", liMode.toString());
}

//
// setAFMode
//
function setAFMode(num mode) {
afMode = mode;
if (afMode == 0) {
buttonAF.setCaption("ALU Forwarding");
showALUForwarding(1);
} else if (afMode == 1) {
buttonAF.setCaption("ALU Interlock");
showALUForwarding(0);
} else if (afMode == 2) {
buttonAF.setCaption("No ALU Interlock");
showALUForwarding(0);
}
setArg("afMode", afMode.toString());
}

//
// setLIMode
//
function setSFMode(num mode) {
sfMode = mode;
if (sfMode == 0) {
buttonSF.setCaption("Store Operand\nForwarding");
showSMDRForwarding(1);
} else if (sfMode == 1) {
buttonSF.setCaption("Store Interlock");
showSMDRForwarding(0);
} else if (sfMode == 2) {
buttonSF.setCaption("No Store Interlock");
showSMDRForwarding(0);
}
setArg("sfMode", sfMode.toString());
}

//
// setCPSRMode
//
function setCPSRMode(num mode) {
zfMode = mode;
if (zfMode == 0) {
buttonZF.setCaption("CPSR Forwarding");
showCPSRForwarding(1);
} else if (zfMode == 1) {
buttonZF.setCaption("CPSR Interlock");
showCPSRForwarding(0);
} else if (zfMode == 2) {
buttonZF.setCaption("No CPSR Interlock");
showCPSRForwarding(0);
}
setArg("zfMode", zfMode.toString());
}

// initialisation

resetCircuit();
```

```
num i, v;
string s;

//
// clear instruction memory
//
for (i = 0; i < 32; i++)
im.setOpcode(4*i, 0);

//
// initialise registers
//
for (i = 0; i < 4; i++) {
s = sprintf("r%d", i);
regFile[i].setValue(getArgAsNum(s, 0));
}

//
// initialise memory
//
for (i = 0; i < 4; i++) {
s = sprintf("m%d", i);
memory[i].setValue(getArgAsNum(s, 0));
}

setTPS(20);
example = getArgAsNum("example", 0);

if (example == 0) {

//
// user defined
//
for (i = 0; i < 32; i++) {
s = sprintf("i%d", i);
im.setOpcode(4*i, getArgAsNum(s, 0));
}

} else if (example == 1) {

//
// Demonstrating SBit Functionality
//
          // SUBS               R1,R2,R3
//
im.setValue( 0, SUB, "S", 0, 1, 2, 3, 0, 0);
im.setValue(4, HALT, " ", 0, 0, 0, 0, 0, 0);
regFile[2].setValue(4);
regFile[3].setValue(4);

} else if (example == 2) {

//
// Demonstrating Condition Functionality
//
          // ADDS               R1,R2,R3
          // SUB                R1,R1,#2
          // B                  L1
          // MOV                R1,#2
          // MOV                R2,#3
          //L1
//
im.setValue( 0, ADD, "S", 0, 1, 2, 3, 0, 0);
im.setValue( 4, SUBi, " ", 0, 1, 1, 2, 0, 0);
im.setValue( 8, B, " ", CS, 0, 0,12, 0, 0);
im.setValue(12, MOVi, " ", 0, 1, 0, 2, 0, 0);
im.setValue(16, MOVi, " ", 0, 2, 0, 3, 0, 0);
im.setValue(20, HALT, " ", 0, 0, 0, 0, 0, 0);
regFile[2].setValue(0xFF);
regFile[3].setValue(0xFF);

} else if (example == 3) {

//
// Demonstrating Condition Functionality
//
          // ADDS               R1,R2,R3
          // SUB                R1,R1,#2
          // MOVCS   R1,#2
          // MOVCS   R2,#3
//
im.setValue( 0, ADD, "S", 0, 1, 2, 3, 0, 0);
im.setValue( 4, SUBi, " ", 0, 1, 1, 2, 0, 0);
im.setValue( 8, MOVi, " ", CS, 1, 0, 2, 0, 0);
im.setValue(12, MOVi, " ", CS, 2, 0, 3, 0, 0);
im.setValue(16, HALT, " ", 0, 0, 0, 0, 0, 0);
regFile[2].setValue(5);
regFile[3].setValue(6);

} else if (example == 4) {

//
// Demonstrating Shifting Functionality
//
          // MOV                R1,#4
//        L1 CMP                R1,#0
          // BLE                HALT
          // LDR                R3,[R0,R1,LSL #2]
          // ADD                R3,R3,R3
          // STR                R3,[R0,R1,LSL #2]
          // B                  L1
//
im.setValue( 0, MOVi, " ", 0, 3, 0, 16, 0, 0);;
im.setValue( 4, STR, " ", 0, 3, 0, 1, LSLi, 2);
im.setValue( 8, HALT, " ", 0, 0, 0, 0, 0, 0);
          regFile[1].setValue(3);
memory[0].setValue(1);
memory[1].setValue(2);
memory[2].setValue(3);
memory[3].setValue(4);


} else if (example == 5) {

          //Factorial
          // Pushing registers 3-1 onto the stack then popping one off

          //im.setValue( 0, JR, 0, 0, 1);
          im.setValue( 0, MOVi, " ", 0, 0, 0, 8, 0, 0);
          im.setValue( 4, STRi, " ", 0, 0, 13, 0xFC, 0, 0);
          im.setValue( 8, SUBi, " ", 0, 13, 13, 4, 0, 0);
          im.setValue(12, B, " ", 0, 0, 0, 0xF0, 0, 0);
          im.setValue(16, HALT, " ", 0, 0, 1, 1, 0, 0);

} else if (example == 6) {

//
// Data Hazard Showing Pipeline Forwarding and Two Stage
//
          // r1 = r3 + r4
          // r3 = r1 - r6                  ;OUT0 Forwarded
          // r5 = r1 + r4 + C   ;OUT1 Forwarded
          // r6 = r1 & r9                  ;Two Stage Clocking Removes need for forwarding
//
im.setValue( 0, ADD, " ", 0, 1, 3, 4, 0, 0);
im.setValue( 4, SUB, " ", 0, 3, 1, 6, 0, 0);
im.setValue( 8, ADC, " ", 0, 5, 1, 4, 0, 0);
im.setValue(12, AND, " ", 0, 6, 1, 9, 0, 0);
im.setValue(16, HALT, " ", 0, 0, 0, 0, 0, 0);
regFile[3].setValue(3);
regFile[4].setValue(4);
regFile[6].setValue(6);
regFile[9].setValue(9);

} else if (example == 7) {

//
// Load Hazard with Poor Scheduling Showing Stall
//
          // r1 = M[r3]
          // r3 = r1 - r6                  ;Stall on this instruction waiting for load to finish
          // r5 = r9 + r4 + C
//
im.setValue( 0, LDRi, " ", 0, 1, 3, 0, 0, 0);
im.setValue( 4, SUB, " ", 0, 3, 1, 6, 0, 0);
im.setValue( 8, ADC, " ", 0, 5, 9, 4, 0, 0);
im.setValue(12, HALT, " ", 0, 0, 0, 0, 0, 0);
regFile[3].setValue(12);
regFile[4].setValue(4);
regFile[6].setValue(6);
regFile[9].setValue(9);

} else if (example == 8) {

//
// Load Hazard with Good Scheduling Showing No Stall
//
           // r1 = M[r3]
           // r5 = r9 + r4 + C
           // r3 = r1 - r6                  ;No Stall on this instruction as load has finished
//
im.setValue( 0, LDRi, " ", 0, 1, 3, 0, 0, 0);
im.setValue( 4, ADC, " ", 0, 5, 9, 4, 0, 0);
im.setValue( 8, SUB, " ", 0, 3, 1, 6, 0, 0);
im.setValue(12, HALT, " ", 0, 0, 0, 0, 0, 0);
regFile[3].setValue(12);
regFile[4].setValue(4);
regFile[6].setValue(6);
regFile[9].setValue(9);

} else if (example == 9) {

           //
// Branch Prediction showing no stall on branches after 1st
//
           //L1 r1 = r3 + r4
           //          r3 = r7 - r6                    ;OUT0 Forwarded
           //          r5 = r9 + r4 + C     ;OUT1 Forwarded
           //          B         L1
//
im.setValue( 0, MOVi, " ", 0, 1, 0, 4, 0, 0);
im.setValue( 4, SUB, " ", 0, 3, 1, 6, 0, 0);
im.setValue( 8, ADC, " ", 0, 5, 1, 4, 0, 0);
im.setValue(12, B, " ", 0, 0, 0, 0xF4, 0, 0);
im.setValue(16, HALT, " ", 0, 0, 0, 0, 0, 0);
regFile[3].setValue(3);
regFile[4].setValue(4);
regFile[6].setValue(6);
regFile[7].setValue(7);
regFile[9].setValue(9);

} else if (example == 10) {

           //Factorial

           //im.setValue( 0, JR, 0, 0, 1);
           im.setValue( 0, MOV, " ", 0, 0, 0, 1, 0, 0);
           im.setValue( 4, CMPi, " ", 0, 0, 1, 1, 0, 0);
           im.setValue( 8, B, " ", LE, 0, 0, 16, 0, 0);
           im.setValue(12, SUBi, " ", 0, 1, 1, 1, 0, 0);
           im.setValue(16, MUL, " ", 0, 0, 1, 0, 0, 0);
           im.setValue(20, B, " ", 0, 0, 0, 0xF0, 0, 0);
           im.setValue(24, HALT, " ", 0, 0, 1, 1, 0, 0);

           regFile[1].setValue(4);

}

//
// save args
//
if (example > 0) {

for (i = 0; i < 32; i++) {
s = sprintf("i%d", i);
setArg(s, im.getOpcode(i * 4).toString());
}

example = (example > maxexample) ? 0 : example

}

num haltOnHalt = getArgAsNum("haltOnHalt", 1);

bpMode = getArgAsNum("bpMode", 0); setBPMode(bpMode);
liMode = getArgAsNum("liMode", 0); setLIMode(liMode);
afMode = getArgAsNum("afMode", 0); setAFMode(afMode);
sfMode = getArgAsNum("sfMode", 0); setSFMode(sfMode);
zfMode = getArgAsNum("zfMode", 0); setCPSRMode(zfMode);
peMode = getArgAsNum("peMode", 0); setPEMode(peMode);
lockCircuit = getArgAsNum("locked", 0);

//
// Help
//
num showHelp = getArgAsNum("help", 1);

Rectangle r = Rectangle2(helpLayer, 0, 0, whiteBrush, 0, 0, WIDTH, HEIGHT);
```

```
r.setOpacity(0.5);
r.setRounded(10, 10);

Pen helpPen = SolidPen(SOLID, 5, RED, ROUND_START | ROUND_JOIN | ROUND_END);
Font helpFont = Font("Arial", 32, BOLD);

Txt(helpLayer, HLEFT | VTOP, 587, 639, helpPen, helpFont, "LEFT CLICK on animation background to start and stop clock.\n\nSHIFT LEFT
CLICK on background to execute \"single ARM clock cycle\".");
if (!lockCircuit) {
r = Rectangle2(helpLayer, 0, helpPen, 0, 72, 152, 176, 25); r.setRounded(5, 5);
r = Rectangle2(helpLayer, 0, helpPen, 0, 55, 424.0, 66, 20); r.setRounded(5, 5);
r = Rectangle2(helpLayer, 0, helpPen, 0, 206, 424.0, 31, 20); r.setRounded(5, 5);
r = Rectangle2(helpLayer, 0, helpPen, 0, 243, 424.0, 31, 20); r.setRounded(5, 5);
r = Rectangle2(helpLayer, 0, helpPen, 0, 280, 424.0, 31, 20); r.setRounded(5, 5);
r = Rectangle2(helpLayer, 0, helpPen, 0, 317, 424.0, 40, 20); r.setRounded(5, 5);
r = Rectangle2(helpLayer, 0, helpPen, 0, 382, 424.0, 31, 20); r.setRounded(5, 5);
Txt(helpLayer, HLEFT | VTOP, 260.0, 135, helpPen, helpFont, "LEFT CLICK to change\ninitial program.");
Txt(helpLayer, HLEFT | VTOP, 425.0, 401, helpPen, helpFont, "LEFT or RIGHT CLICK to \"rotate\"\ninstructions and operands.\nHold and
release to reset value.");
}

r = Rectangle2(helpLayer, 0, helpPen, 0, 983, 90, 32, 64);
r.setRounded(10, 10);
Txt(helpLayer, HLEFT | VTOP, 1025.0, 88, helpPen, helpFont, "LEFT or RIGHT CLICK register\nto increment or decrement value.");

if (!lockCircuit) {
r = Rectangle2(helpLayer, 0, helpPen, 0, 242, 1330.0, 1212, 30);
r.setRounded(10, 10);
Txt(helpLayer, HLEFT | VTOP, 587.0, 959, helpPen, helpFont, "LEFT CLICK on any of the buttons below to change circuit
configuration.");
}

Rectangle closeHelp = Rectangle2(helpLayer, 0, helpPen, yellowBrush, 925.0, 507, 270, 66, helpPen, helpFont, "CLOSE HELP");
closeHelp.setRounded(5, 5);

when closeHelp ~> eventEE(num enter, num x, num y) {
closeHelp.setBrush(enter ? gray224Brush : yellowBrush);
}

when closeHelp ~> eventMB(num down, num flags, num x, num y) {
           if (down && (flags & MB_LEFT)) {
                      setArg("help", "0");
                      helpLayer.setOpacity(0);
           }
           return 0;
}

if (showHelp == 0)
helpLayer.setOpacity(0);

// eof


//
// stackpointer.vin
//
// Simulation of the ARM
// Written by Rónán Dowling-Cullen, Final Year Computer Science 2018/19DLX
// Building on the simulation of the DLX written by Edsko de Vries, Summer 2003

class StackPointer(num _x1, num _y1, num _x2, num _y2, num w, num _regWidth) {

           num x1 = _x1, y1 = _y1, x2 = _x2, y2 = _y2, regWidth = _regWidth;
Pen fgPen0 = SolidPen(SOLID, w, RED, BEVEL_JOIN | BUTT_END);
Pen fgPen1 = SolidPen(SOLID, w, RED, BEVEL_JOIN | ARROW60_END);

           Line bodies[8];
           Line heads[8];
           bodies[0] = Line2(0, ABSOLUTE, fgPen0, x1, y1-1*regWidth, x2-w, y2-1*regWidth);
           bodies[1] = Line2(0, ABSOLUTE, fgPen0, x1, y1+0*regWidth, x2-w, y2+0*regWidth);
           bodies[2] = Line2(0, ABSOLUTE, fgPen0, x1, y1+1*regWidth, x2-w, y2+1*regWidth);
           bodies[3] = Line2(0, ABSOLUTE, fgPen0, x1, y1+2*regWidth, x2-w, y2+2*regWidth);
           bodies[4] = Line2(0, ABSOLUTE, fgPen0, x1, y1+3*regWidth, x2-w, y2+3*regWidth);
           bodies[5] = Line2(0, ABSOLUTE, fgPen0, x1, y1+4*regWidth, x2-w, y2+4*regWidth);
           bodies[6] = Line2(0, ABSOLUTE, fgPen0, x1, y1+5*regWidth, x2-w, y2+5*regWidth);
           bodies[7] = Line2(0, ABSOLUTE, fgPen0, x1, y1+6*regWidth, x2-w, y2+6*regWidth);

           heads[0] = Line2(0, ABSOLUTE, fgPen1, x2-w, y2-1*regWidth, x2, y2-1*regWidth);
           heads[1] = Line2(0, ABSOLUTE, fgPen1, x2-w, y2+0*regWidth, x2, y2+0*regWidth);
           heads[2] = Line2(0, ABSOLUTE, fgPen1, x2-w, y2+1*regWidth, x2, y2+1*regWidth);
           heads[3] = Line2(0, ABSOLUTE, fgPen1, x2-w, y2+2*regWidth, x2, y2+2*regWidth);
           heads[4] = Line2(0, ABSOLUTE, fgPen1, x2-w, y2+3*regWidth, x2, y2+3*regWidth);
           heads[5] = Line2(0, ABSOLUTE, fgPen1, x2-w, y2+4*regWidth, x2, y2+4*regWidth);
```

```
            heads[6] = Line2(0, ABSOLUTE, fgPen1, x2-w, y2+5*regWidth, x2, y2+5*regWidth);
            heads[7] = Line2(0, ABSOLUTE, fgPen1, x2-w, y2+6*regWidth, x2, y2+6*regWidth);

            setPosOpacity(1,0);
            setPosOpacity(2,0);
            setPosOpacity(3,0);
            setPosOpacity(4,0);
            setPosOpacity(5,0);
            setPosOpacity(6,0);
            setPosOpacity(7,0);
            setPosOpacity(0,1);


            function setPosOpacity(num position, num opacity){
                        setOpacity(0);
                        bodies[position].setOpacity(opacity);
                        heads[position].setOpacity(opacity);
            }

            function setOpacity(num opacity) {
                        num i = 0;
                        while(i < 8){
                                    bodies[i].setOpacity(0);
                                    heads[i].setOpacity(0);
                                    i++;
                        }
            }

            function moveSP(num sp){
                        num position = ((0x40 - sp)/4);
                        setPosOpacity(position,1);
            }


}

// eof
```

A3 Python Scaling Script

```python
#!/usr/bin/python

import sys
import re

print('Number of arguments:', len(sys.argv), 'arguments.')
print('Argument List:', str(sys.argv))

args = sys.argv
ifil = args[1]
oldWidth = float(args[2])
newWidth = float(args[3])
oldHeight = float(args[4])
newHeight = float(args[5])


with open(ifil) as infile, open('./testo.txt', 'w') as outfile:
    for i,line in enumerate(infile):
        # Ignoring comments
        if not re.match('\/\/.*',line):
            # Pattern matching all Line2 constructors
            matchObj = re.match('.*Line2\(.+,.+,.+, ?(\d+) ?, ?(\d+) ?, ?(\d+) ?, ?(\d+) ?, ?(\d+)
?, ?(\d+) ?, ?(\d+) ?, ?(\d+) ?\)',line)
            if matchObj:
                newNums = []
                newNums.append((float(matchObj.group(1))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(2))/oldHeight)*newHeight)
                newNums.append((float(matchObj.group(3))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(4))/oldHeight)*newHeight)
                newNums.append((float(matchObj.group(5))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(6))/oldHeight)*newHeight)
                newNums.append((float(matchObj.group(7))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(8))/oldHeight)*newHeight)
                for i in range(0,8):
                    line = line.replace(matchObj.group(i+1),str(round(newNums[i])))
            elif re.match('.*Line2\(.*\)',line):
                matchObj = re.match('.*Line2\(.+,.+,.+, ?(\d+) ?, ?(\d+) ?, ?(\d+) ?, ?(\d+) ?,
?(\d+) ?, ?(\d+) ?\)',line)
                if matchObj:
                    newNums = []
                    newNums.append((float(matchObj.group(1))/oldWidth)*newWidth)
                    newNums.append((float(matchObj.group(2))/oldHeight)*newHeight)
                    newNums.append((float(matchObj.group(3))/oldWidth)*newWidth)
                    newNums.append((float(matchObj.group(4))/oldHeight)*newHeight)
                    newNums.append((float(matchObj.group(5))/oldWidth)*newWidth)
                    newNums.append((float(matchObj.group(6))/oldHeight)*newHeight)
                    for i in range(0,6):
                        line = line.replace(matchObj.group(i+1),str(round(newNums[i])))
                elif re.match('.*Line2\(.*\)',line):
                    matchObj = re.match('.*Line2\([\s\S\d]* ?, ?[\s\S\d]* ?, ?[\s\S\d]* ?, ?(\d+) ?,
?(\d+) ?, ?(\d+) ?, ?(\d+) ?\)',line)
                    if matchObj:
                        newNums = []
                        newNums.append((float(matchObj.group(1))/oldWidth)*newWidth)
                        newNums.append((float(matchObj.group(2))/oldHeight)*newHeight)
                        newNums.append((float(matchObj.group(3))/oldWidth)*newWidth)
                        newNums.append((float(matchObj.group(4))/oldHeight)*newHeight)
                        for i in range(0,4):
                            line = line.replace(matchObj.group(i+1),str(round(newNums[i])))
                    elif re.match('.*Line2\(.*\)',line):
                        print("Manual update needed at line " + str(i+1) + " for Line2 call.")



            # Pattern matching all Font constructors
            matchObj = re.match('.*Font\([\s\S\d]* ?, ?(\d+) ?,.*\)',line)
            if matchObj:
                old = oldWidth
                new = newWidth
                if (newHeight/oldHeight) > (newWidth/oldWidth):
                    old = oldHeight
                    new = newHeight
                newNums = []
                newNums.append((float(matchObj.group(1))/old)*new)
                for i in range(0,1):
                    line = line.replace(matchObj.group(i+1),str(round(newNums[i])))
            elif re.match('.*Font\(.*\)',line):
                print("Manual update needed at line " + str(i+1) + " for Font call.")


            # Pattern matching all Rectangle2 constructors
            matchObj = re.match('.*Rectangle2\(.+, ?(\d+) ?, ?(\d+) ?, ?(\d+) ?, ?(\d+) ?,
?(\d+).*\)',line)
            if matchObj:
                newNums = []
                newNums.append((float(matchObj.group(1))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(2))/oldHeight)*newHeight)
                newNums.append((float(matchObj.group(3))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(4))/oldHeight)*newHeight)
                for i in range(0,4):
                    line = line.replace(matchObj.group(i+1),str(round(newNums[i])))
            elif re.match('.*Rectangle2\(.*\)',line):
                print("Manual update needed at line " + str(i+1) + " for Rectangle2 call.")


            # Pattern matching all SolidPen constructors
            matchObj = re.match('.*SolidPen\(.+, ?(\d+).*\)',line)
            if matchObj:
                old = oldWidth
                new = newWidth
                if (newHeight/oldHeight) > (newWidth/oldWidth):
                    old = oldHeight
                    new = newHeight
                newNums = []
                newNums.append((float(matchObj.group(1))/old)*new)
                for i in range(0,1):
                    line = line.replace(matchObj.group(i+1),str(round(newNums[i])))
            elif re.match('.*SolidPen\(.*\)',line):
                print("Manual update needed at line " + str(i+1) + " for SolidPen call.")


            # Pattern matching all Button constructors
            matchObj = re.match('.*Button\( ?(\d+) ?, ?(\d+) ?, ?(\d+) ?, ?(\d+) ?.*\)',line)
            if matchObj:
                newNums = []
                newNums.append((float(matchObj.group(1))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(2))/oldHeight)*newHeight)
                newNums.append((float(matchObj.group(3))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(4))/oldHeight)*newHeight)
                for i in range(0,4):
                    line = line.replace(matchObj.group(i+1),str(round(newNums[i])))
            elif re.match('.*Button\(.*\)',line):
                print("Manual update needed at line " + str(i+1) + " for Button call.")

            # Pattern matching all Image constructors
            matchObj = re.match('.*Image\( ?.+ ?, ?.+ ?, ?.+ ?,.+, ?(\d+) ?, ?(\d+) ?, ?(\d+) ?,
?(\d+) ?, ?(\d+) ?, ?(\d+) ?.*\)',line)
            if matchObj:
                newNums = []
                newNums.append((float(matchObj.group(1))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(2))/oldHeight)*newHeight)
                newNums.append((float(matchObj.group(3))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(4))/oldHeight)*newHeight)
                newNums.append((float(matchObj.group(5))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(6))/oldHeight)*newHeight)
                for i in range(0,9):
                    line = line.replace(matchObj.group(i+1),str(round(newNums[i])))
            elif re.match('.*Image\(.*\)',line):
                print("Manual update needed at line " + str(i+1) + " for Image call.")

            # Txt(0, HLEFT | VTOP, 10, 46, darkGrayPen, font, "instructions executed:");
            matchObj = re.match('.*Txt\(.*[a-zA-Z], ?(\d+) ?, ?(\d+).*\)',line)
            if matchObj:
                newNums = []
                newNums.append((float(matchObj.group(1))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(2))/oldHeight)*newHeight)
                for i in range(0,2):
                    line = line.replace(matchObj.group(i+1),str(round(newNums[i])))
            elif re.match('.*Txt\(.*\)',line):
                print("Manual update needed at line " + str(i+1) + " for Txt call.")


            # InstructionMemory(10, 80, 100, 320);
            matchObj = re.match('.*InstructionMemory\( ?(\d+) ?, ?(\d+) ?, ?(\d+) ?,
?(\d+).*\)',line)
            if matchObj:
                newNums = []
                newNums.append((float(matchObj.group(1))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(2))/oldHeight)*newHeight)
                newNums.append((float(matchObj.group(3))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(4))/oldHeight)*newHeight)
                for i in range(0,4):
                    line = line.replace(matchObj.group(i+1),str(round(newNums[i])))
            elif re.match('.*InstructionMemory\(.*\)',line):
                print("Manual update needed at line " + str(i+1) + " for InstructionMemory call.")

            # AnimatedClock(20, 410, 80, 30);
            matchObj = re.match('.*AnimatedClock\( ?(\d+) ?, ?(\d+) ?, ?(\d+) ?, ?(\d+).*\)',line)
            if matchObj:
                newNums = []
                newNums.append((float(matchObj.group(1))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(2))/oldHeight)*newHeight)
                newNums.append((float(matchObj.group(3))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(4))/oldHeight)*newHeight)
                for i in range(0,4):
                    line = line.replace(matchObj.group(i+1),str(round(newNums[i])))
            elif re.match('.*AnimatedClock\(.*\)',line):
                print("Manual update needed at line " + str(i+1) + " for AnimatedClock call.")

            # Register(200, 210, 20, 40, TOP, "PC");
```

```
            matchObj = re.match('.*Register\( ?(\d+) ?, ?(\d+) ?, ?(\d+) ?, ?(\d+).*\)',line)
            if matchObj:
                newNums = []
                newNums.append((float(matchObj.group(1))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(2))/oldHeight)*newHeight)
                newNums.append((float(matchObj.group(3))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(4))/oldHeight)*newHeight)
                for i in range(0,4):
                    line = line.replace(matchObj.group(i+1),str(round(newNums[i])))
            elif re.match('.*Register\(.*\)',line):
                print("Manual update needed at line " + str(i+1) + " for
Register/InstructionRegister call.")

            # Component(200, 170, 30, 10, "mux 2");
            matchObj = re.match('.*Component\( ?(\d+) ?, ?(\d+) ?, ?(\d+) ?, ?(\d+).*\)',line)
            if matchObj:
                newNums = []
                newNums.append((float(matchObj.group(1))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(2))/oldHeight)*newHeight)
                newNums.append((float(matchObj.group(3))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(4))/oldHeight)*newHeight)
                for i in range(0,4):
                    line = line.replace(matchObj.group(i+1),str(round(newNums[i])))
            elif re.match('.*Component\(.*\)',line):
                print("Manual update needed at line " + str(i+1) + " for Component call.")

            # .addPoint(110, 390);
            matchObj = re.match('.*.addPoint\( ?(\d+) ?, ?(\d+).*\)',line)
            if matchObj:
                newNums = []
                newNums.append((float(matchObj.group(1))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(2))/oldHeight)*newHeight)
                for i in range(0,2):
                    line = line.replace(matchObj.group(i+1),str(round(newNums[i])))
            elif re.match('.*.addPoint\(.*\)',line):
                print("Manual update needed at line " + str(i+1) + " for .addPoint call.")

            # Rectangle(activePipesLayer, 0, 0, redBrush, 180,390, -30,-6, 60,12, whitePen, font);
            matchObj = re.match('.*Rectangle\(.+,.+,.+,.+, ?(\d+) ?, ?(\d+) ?, ?-?(\d+) ?, ?-?(\d+)
?, ?(\d+) ?, ?(\d+) ?.*\)',line)
            if matchObj:
                newNums = []
                newNums.append((float(matchObj.group(1))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(2))/oldHeight)*newHeight)
                newNums.append((float(matchObj.group(3))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(4))/oldHeight)*newHeight)
                newNums.append((float(matchObj.group(5))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(6))/oldHeight)*newHeight)
                for i in range(0,6):
                    line = line.replace(matchObj.group(i+1),str(round(newNums[i])))
            elif re.match('.*Rectangle\(.*\)',line):
                print("Manual update needed at line " + str(i+1) + " for Rectangle call.")

            # Line(valuesLayer, ABSOLUTE, controlPen, 0,0, 370,135, 345,135, 345,160);
            matchObj = re.match('.*Line\(.+,.+,.+, ?(\d+) ?, ?(\d+) ?, ?(\d+) ?, ?(\d+) ?, ?(\d+) ?,
?(\d+) ?, ?(\d+) ?, ?(\d+) ?.*\)',line)
            if matchObj:
                newNums = []
                newNums.append((float(matchObj.group(1))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(2))/oldHeight)*newHeight)
                newNums.append((float(matchObj.group(3))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(4))/oldHeight)*newHeight)
                newNums.append((float(matchObj.group(5))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(6))/oldHeight)*newHeight)
                newNums.append((float(matchObj.group(7))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(8))/oldHeight)*newHeight)
                for i in range(0,8):
                    line = line.replace(matchObj.group(i+1),str(round(newNums[i])))
            elif re.match('.*Line\(.*\)',line):
                print("Manual update needed at line " + str(i+1) + " for Line call.")

            # .setPoint(0, 420, 205);
            matchObj = re.match('.*.setPoint\(.+, ?-?(\d+) ?, ?-?(\d+).*\)',line)
            if matchObj:
                newNums = []
                newNums.append((float(matchObj.group(1))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(2))/oldHeight)*newHeight)
                for i in range(0,2):
                    line = line.replace(matchObj.group(i+1),str(round(newNums[i])))
            elif re.match('.*.setPoint\(.*\)',line):
                print("Manual update needed at line " + str(i+1) + " for .setPoint call.")

            # .setPt(0, 420, 205);
            matchObj = re.match('.*.setPt\(.+, ?-?(\d+) ?, ?-?(\d+).*\)',line)
            if matchObj:
                newNums = []
                newNums.append((float(matchObj.group(1))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(2))/oldHeight)*newHeight)
                for i in range(0,2):
                    line = line.replace(matchObj.group(i+1),str(round(newNums[i])))
            elif re.match('.*.setPt\(.*\)',line):
                print("Manual update needed at line " + str(i+1) + " for .setPt call.")

            # ALU(490, 190, 40, 80);
            matchObj = re.match('.*ALU\( ?(\d+) ?, ?(\d+) ?, ?(\d+) ?, ?(\d+).*\)',line)
            if matchObj:
                newNums = []
                newNums.append((float(matchObj.group(1))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(2))/oldHeight)*newHeight)
                newNums.append((float(matchObj.group(3))/oldWidth)*newWidth)
                newNums.append((float(matchObj.group(4))/oldHeight)*newHeight)
                for i in range(0,4):
                    line = line.replace(matchObj.group(i+1),str(round(newNums[i])))
            elif re.match('.*ALU\(.*\)',line):
                print("Manual update needed at line " + str(i+1) + " for ALU call.")

        outfile.write(line)

print("\nScaled version of " + ifil + " generated in testo.txt")
```

A4 Python Shifting Script

```python
#!/usr/bin/python

import sys
import re

print('Number of arguments:', len(sys.argv), 'arguments.')
print('Argument List:', str(sys.argv))

args = sys.argv
ifil = args[1]
minX = float(args[2])
shiftX = float(args[3])


with open(ifil) as infile, open('./testo.txt', 'w') as outfile:
    for i,line in enumerate(infile):
        # Ignoring comments
        if not re.match('\/\/.*',line):
            # .setPoint(0, 420, 205);
            matchObj = re.match('.*.setPoint\(.+, ?-?(\d+) ?, ?-?(\d+).*\)',line)
            if matchObj:
                num = int(matchObj.group(1))
                if(num > minX): line = line.replace(str(num),str(num+shiftX))
            elif re.match('.*.setPoint\(.*\)',line):
                print("Manual update needed at line " + str(i+1) + " for .setPoint call.")

            # .setPt(0, 420, 205);
            matchObj = re.match('.*.setPt\(.+, ?-?(\d+) ?, ?-?(\d+).*\)',line)
            if matchObj:
                num = int(matchObj.group(1))
                if(num > minX): line = line.replace(str(num),str(num+shiftX))
            elif re.match('.*.setPt\(.*\)',line):
                print("Manual update needed at line " + str(i+1) + " for .setPt call.")

            # Match non setPoint
            numTups = re.findall('\d+ ?, ?\d+',line)
            i = 0;
            while i < len(numTups):
                nums = [int(s) for s in re.findall('\d+',numTups[i])]
                if nums[0] > minX:
                    line = line.replace(str(nums[0]),str(nums[0]+shiftX))
                i+=1
        outfile.write(line)


print("\Shifted version of " + ifil + " generated in testo.txt")
```