M2 Architecture And Implementation

Zakary Nafziger

15 December 2017

Contents

List of Tables

List of Figures

1 Introduction

The M2CPU is a simple subscalar 8-bit processor that implements the 75 instruction M2 instruction set. The M2CPU and M2 instruction set were designed and implemented by the author as a personal hobby project. The M2 architecture is intended to be as simple as possible while still being interesting. Some basic information:

- 8-bit data and ALU buses
- 16-bit address bus; 64KB of addressable memory
- 256B zero-page stack
- 75 instructions
- implemented in VHDL on a MAX10 based development board

This document describes the M2CPU. The M2CPU is only one part of a larger project, the M2 computer. As of this writing the other major components of the M2 computer project are an assembler and VGA video system. The assembler and video system are documented separately.

1.1 Terminology

Throughout this document the terms M2CPU, M2 instruction set, and M2 architecture will be used. M2CPU refers to the VHDL processor that implements the M2 instruction set. The M2 instruction set is the set of 75 instructions that make up the M2 assembly language. The M2 architecture refers to the combination of a processor and instruction set that implement the M2 assembly language. The M2CPU is therefore an implementation of the M2 architecture, since it supports the complete M2 instruction set. However since the M2CPU's instructions are implemented in microcode it could implement a different architecture. This document describes a complete implementation of the M2 architecture, namely the M2CPU.

The name 'M2' stands for 'Model 2' The 'Model 1' processor was a 4-bit system that was constructed by the Author. The Model 1 was implemented in discrete hardware, (about 1200 resistors and transistors). The Model 1 is documented here: `https://hackaday.io/project/665-4-bit-computer-built-from-discrete-transistors`

2 Instruction Set

The M2CPU implements the M2 instruction set. The M2 instruction set is a custom instruction set designed for this project. The instruction set consists of 75 instructions that can be broadly grouped into four categories: register manipulation, memory access, arithmetic and logical, and program control.

2.1 Register Manipulation Instructions

The register manipulation instructions allow data to be loaded from one register to another. The Register manipulation instructions allow completely orthogonal movement of data between general purpose registers (GPRs), restricted data movement between special purpose registers (SPRs) and GPRs, and no data movement between two SPRs. Most of the M2CPU's registers are 8-bits wide, however the addressing registers may be manipulated 16 bits at a time. Each GPR serves a multiple purposes. The primary purpose of the GPRs is to provide fast data storage for a program. The secondary purpose(s) of each GPR is as the target or source of data for a given operation. Each SPR serves a single purpose. The Data held in an SPR is always interpreted in a consistent manner. The following subsections provide a detailed description of each register's function:

2.1.1 A Register

The `A` register is an 8-bit GPR. The secondary function of the `A` register is as the destination of ALU operations. The result of an ALU operation is always stored in the `A` register. The designation '`A`' is meant to evoke 'accumulator'.

2.1.2 G Register

The `G` register is an 8-bit GPR. The `G` register's secondary function is to hold the high byte of an address. Some memory instructions are implicitly addressed and since the M2CPU supports a 64KB address space addresses are 16-bits wide. The low byte of an implicit address is stored in the `H` register. Some register manipulation instructions act on the `G` register individually while others treat the combined `GH` register as a single 16-bit unit.

2.1.3 H Register

The `H` register is an 8-bit GPR. The `H` register's secondary function is as the low byte of an implicit address as described in the previous section.

2.1.4 X Register

The `X` register is an 8-bit GPR. The `X` register serves two secondary purposes. The first is as an ALU operand, and the second is as the point of access to the Stack Pointer. Since the Stack Pointer is an SPR it's contents can only be accessed through a GPR, the `X` register is that GPR.

2.1.5 Y register

The `Y` register is an 8-bit GPR. The `Y` register also serves two secondary purposes. The first is as the an ALU operand and the second is as a point of access to the Status register. In general the `Y` register serves a similar purpose as the `X` register.

2.1.6 Status Register

The Status Register (`S` register) is an 8-bit SPR. The low four bits of the `S` register hold the processor's status flags. The status flags are automatically manipulated by the ALU after every arithmetic or logical operation and may also be manipulated by some control instructions. The status flags are used by the control instructions to control program flow. The contents of the `S` register can be accessed through the `Y` register. The status flags are:

- Zero (`z`, bit 3). Set when the last ALU operation produced zero as a result
- Negative (`n`, bit 2). Set when the result of the last ALU operations was negative (as interpreted in two's complement)
- Carry (`c`, bit 1). Set when the result of the last ALU operation produced a '9th bit', i.e. unsigned addition of two numbers greater than 127.
- Overflow (`o`, bit 0). Set when the result of the last ALU operation produced an impossible result (when interpreted in two's complement)

Status flags are positive logic, i.e. set is '1' and clear is '0'.

2.1.7 Stack Pointer

The Stack Pointer (SP) is an 8-bit SPR. The SP holds the address of the top of the stack. All stack addresses are mapped onto the zeroth page of memory and can therefore be represented with a single byte. The SP starts at address 0x0000 and is incremented after each push. This means that the stack grows from low addresses to high addresses. The value of the SP can be accessed through the X register.

2.1.8 Program Counter

The Program Counter (PC) is a 16-bit SPR. The PC holds the current memory address. The PC is incremented after each machine cycle so that the next instruction can be fetched. Some instructions may load the PC with a value other than the next instruction. The PC can be accessed through the combined GH register.

2.2 Memory Manipulation Instructions

Memory manipulation instructions are instructions that move data from memory to a register or from a register to memory. No other category of instruction manipulates values in memory. There are three types of memory manipulation instruction: direct access, literal to register, and stack operations.

2.2.1 Direct Access

Direct Access instructions move a single byte from memory to a given GPR or from a given GPR to memory. The address of the byte to manipulate is held in the combined GH register. From a machine language perspective direct access instructions look like register manipulation instructions with the added caveat that the value of the 'M' register is dependant on the address held in the GH register. In general direct access instructions take about twice as long as register manipulation instructions.

2.2.2 Literal to Register

Literal to register instructions load a single byte literal in to a given GPR. Literal to register instructions are similar to direct access instructions, except that no address is required. The byte to be loaded is stored in the address immediately following the address storing the instruction code. This means that these instructions take two addresses when assembled.

2.2.3 Stack Operations

Stack operations place a byte on the stack (push) or remove a byte from the stack (pop) and adjust the `SP` as necessary. Bytes are pushed from or popped to GPRs. The stack is stored in the first page of memory; addresses `0x0000` to `0x00FF`. The `SP` wraps without warning at the edges of the stack. This means that stack safety is the sole responsibility of the programmer. The M2CPU will happily increment the `SP` from `0xFF` to `0x00` if a byte is pushed to an already full stack without any type of warning. This unsafe behaviour was implemented to keep stack instruction complexity to a minimum. As it stands stack operations are the most complex and slow instructions in the M2 instruction set.

2.3 Arithmetic and Logical Instructions

Arithmetic and logical instructions preform an arithmetic or logical function on the `X` and `Y` registers, store the result in the `A` register and update the flags in the `S` register. The M2CPU's ALU can calculate 8 different functions (detailed in section 2.6). Six of these functions require two inputs (`X` and `Y`), and two functions require only one input, (`X`).

2.4 Program Control Instructions

Program control instructions control the flow of the program. There are four types of program control instructions: no-operation, unconditional jump, conditional branch, and status flag manipulation.

As the name suggests the no-operation does nothing. The no-operation is not quite a program control instruction, it is more accurately 'the lack of an instruction'. The behaviour of the no-operation is well defined so that if a

program attempts to execute uninitialized memory the processor will behave in a predictable (and safe) manner. A no-operation takes an entire machine cycle and one microstate to execute. More information about machine cycles and microstates is found in sections 3.3 and 4 respectively.

2.4.1 Unconditional Jump

An unconditional jump sets the address of the next instruction to be executed to the address held in the `GH` register. Since the `PC` is incremented at the end of every machine cycle (see section 4.1) the jump instruction must load the `PC` with one less than the value stored in the `GH` register. so that the next instruction executed is the one held at the address `GH`.

2.4.2 Conditional Branch

Conditional branch instructions change which instruction is executed next based on the value of a status flag. In general conditional branch instructions allow the next instruction to be executed if some condition is not met. If the condition is met the next instruction is skipped and the instruction following it is executed next. This means that the two branches differ by exactly one instruction. The condition evaluated is the state of one of the status flags (section 2.1.6). Since there are four status flags, and each can have two values (set or clear) there are eight conditional branch instructions, i.e. branch if flag set, and branch if flag clear. This branching mechanism is somewhat austere, however it serves to reduce the overall complexity of the M2CPU.

Some useful assembly language idioms are the 'escape jump' and 'if-else'. Since two branches may only differ by one instruction it is often useful to make that one instruction an unconditional jump. For example:

```
LDG 0x0A # load the high byte of
         # the escape address 0x0AA0
LDH 0xA0 # load the low byte of 0x0AA0
BZC # branch if zero is clear
JMP # executed if zero is set
LDX A # executed if zero is clear
.
.
.
# address 0x0AA0
```

```
LDY A # this code only runs if the
      # zero flag was set before the branch.
```

An 'if-else' statement can be implemented as follows:

```
LDX 0x10 # put some data in the X
         # and Y registers
LDY 0x01
BZS
LDM X # this instruction is executed
      # if the zero flag is clear
BZC
LDM Y # this instruction is executed
      # if  the zero flag is set
...
# memory is loaded with X if zero is
# clear, else memory is loaded with Y
```

2.4.3 Status Flag Manipulation

Status flag manipulation instructions allow the programmer to set the value of any of the four status flags. There are 8 status flag manipulation instructions since each flag can be set ('1') or cleared ('0'). The status flag manipulation instructions are the only way to move data into the `S` register. The contents of the `S` register may be dumped into the `Y` register. This allows status flags to be stored, or inspected by the program.

3 M2CPU Architecture

This section outlines the modular architecture of the M2CPU. The implementation of this architecture is described in section 5. The M2CPU is designed (and implemented) as a series of modules that communicate over three major buses. The way that these modules interact with each other to carry out instructions is described in the next section (section 4). In general This section describes the processor's 'hardware', whereas section 4 describes how this 'hardware' can be made to implement an instruction set.

3.1 Block Diagram

The architecture of the M2CPU is well described by a block diagram. Figure 1 shows all of the major components of the processor and is organized very similarly to the actual VHDL implementation of the M2CPU (section 5). The dotted line in figure 1 denotes the boundary of the processor and serves to show the separation between the processor and memory. The dashed arrows denote single bit control buses (section 4), the light weight arrows denote 8-bit data and instruction buses, and the medium weight arrows denote 16-bit address buses. The heavy weight line denotes the control bus. The bus architecture is discussed in further detail in the next section (3.2).

The M2CPU's internal structure can be loosely divided into two halves; the processing logic and the control logic. Roughly speaking processing logic is on the left half of figure 1 and control logic is on the right half. In general programs manipulate information with the processing logic. As such the functionality of the processing logic is exposed to the programmer through the instruction set (section 2). The control logic is responsible for actually running a program. The primary functions of the control logic are to fetch and decode instructions in sequence. Since the operation of the control logic is largely invisible to the programmer it is described in detail in section 4.

In an effort to preserve the legibility of figure 1 a number of abbreviations have been used to label the processor's functional blocks. These abbreviations are described below:

- Processing Logic:
  - `A`: The `A` register.

  - `G`, and `H`: The `G` register and `H` register. Note that these registers are grouped. This is reflective of how some instructions treat the `G` and `H` registers as a single 16-bit `GH` register.
  - `X`: The `X` register.
  - `Y`: The `Y` register.
  - `S`: The status register.
  - `SP`: The stack pointer.
  - `PC`: The program counter. Note that the program counter is twice as wide a the other registers. This is reflective of the fact that the program counter holds a 16-bit address instead of a single byte.
  - `ALU`: The arithmetic and logic unit.
- Control Logic:
  - `ABM`: Address bus multiplexer
  - `DBM`: Data bus multiplexer
  - `IR`: Instruction register
  - `FSM`: Finite state machine
  - `CBEN`: Control bus enable
  - `MICROCODE`: The microcode look up table
  - `DECODE`: Branch decoding logic
- `RST` and `CLK`: The processor's reset and clock lines respectively.

A detailed description of each functional block and how it maps to actual VHDL is given in section 5.

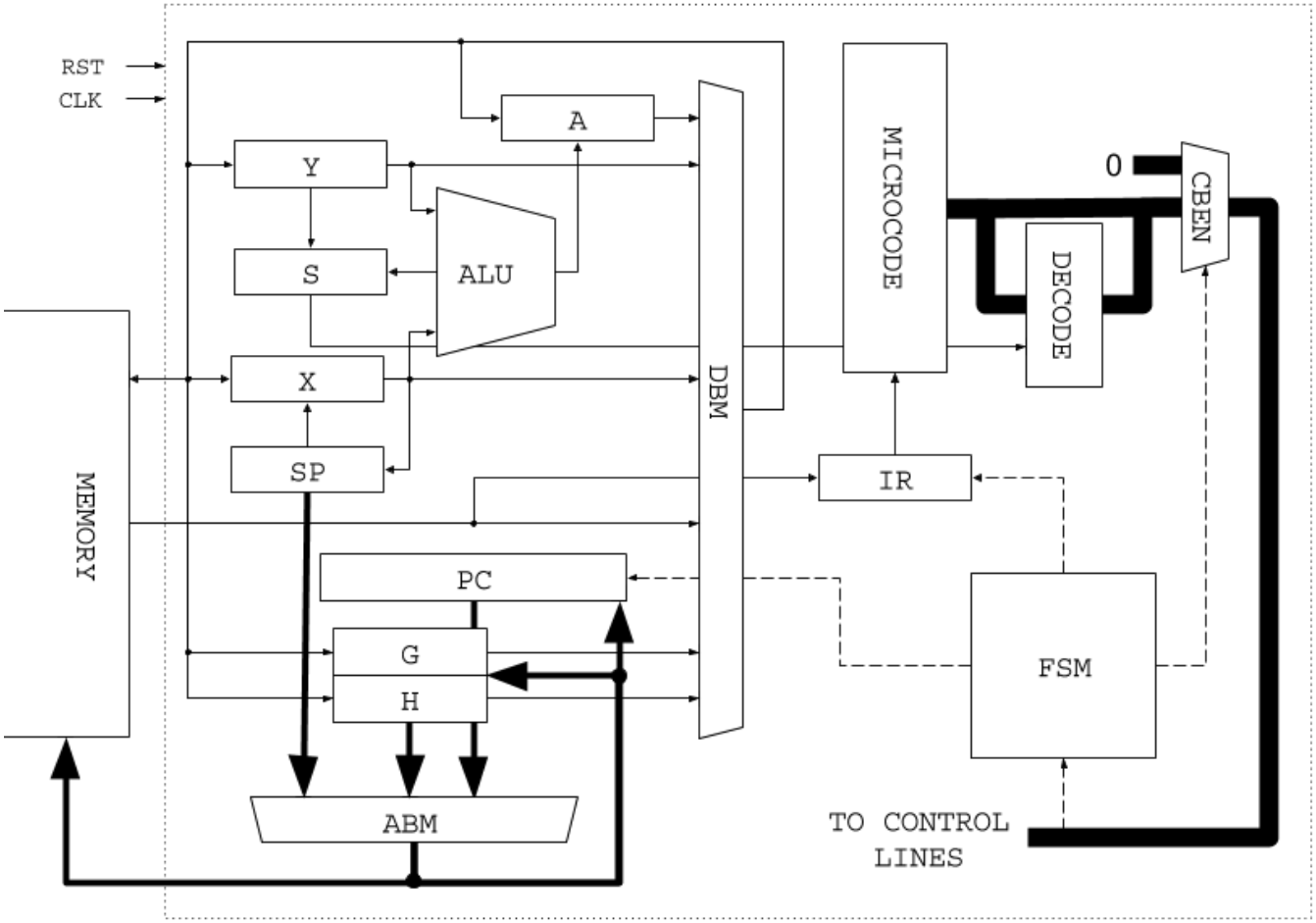

Figure 1: Simplified M2CPU Block Diagram

3.2 Bus Architecture

The M2CPU is designed around 3 major buses; the 8-bit data bus, the 16-bit address bus and the 42-bit control bus. The data bus is used to move data around the processor. Bytes on the data bus are never interpreted as instructions, however instructions can be placed on the data bus if the programmer desires. The address bus is used to move addresses around the processor. The processor's 8-bit data buses are strictly separated from the 16-bit address buses. The only points of access between the 16-bit bus and the 8-bit bus are the stack pointer and the combined `GH` register. The control bus is the collection of control signals coming from the processing logic. The control bus is made up of load, increment, and select lines. The implementation of the M2CPU's bus architecture is described in section 5.

4 Control Logic

The control logic is responsible for manipulating the processing logic in such a way that an instruction is carried out. For any given instruction a variety of low level operations must be carried out. The instruction must be fetched from a memory address, it must be decoded, control signals must be applied to the control bus for specific lengths of time, and finally the address of the next instruction must be calculated. This general sequence of operations is the machine cycle. The specifics of the M2CPU's machine cycle are discussed in the next section (4.1).

For each instruction some stages of the machine cycle are the same. All instructions must be fetched and decoded, and the address of the next instruction must be determined before the next machine cycle can begin. However every instruction does a different thing. In the M2CPU the stages of the machine cycle that differ for each instruction are written in microcode. This means that some portion of each machine cycle is spent executing instruction specific microcode rather than instruction agnostic machine states.

Some instructions take more time to execute than others. This means that the machine cycle for one instruction may have a different number of microcode steps than the machine cycle for another. The M2CPU allows an arbitrary number of microcode steps to be executed for each instruction. This is one of the few optimizations made in the M2 architecture. This optimization was deemed necessary since some instructions in the M2 instruction set differ in execution time by 300%. If this optimization was not made and the M2CPU had a fixed length machine cycle the fastest (and most used) instructions would leave the processor idle for around two thirds of its execution time.

4.1 Machine Cycle

The machine cycle is the series of steps that are carried out to execute an instruction. In the M2CPU the machine cycle is controlled by a finite state machine (FSM). A state diagram for the M2CPU's FSM is shown in figure 2. The FSM has six states. Transitions between states occur on the rising edge of the system clock. To accommodate a variable number of microcode steps the M2CPU may remain in the `EXEC` state for an arbitrary number of clock cycles.

The processor resets to the SETUP state, as such SETUP is the first state in the machine cycle. In the SETUP state the value held in the program counter is placed on the address bus (via the ABM, section 3.2). Additionally the control bus is set to zero (via the CBEN). This state ensures that the address of the next instruction to execute is placed on the address bus and is not changing by the time memory access starts. This state sets up the address so that memory access can begin.

The second and third states in the machine cycle (FETCH_1 and FETCH_2) allow time for the next instruction to be retrieved from memory. Memory requires two clock cycles to access, one to latch the address in and one to latch data out. Since the program counter's value cannot be guaranteed until the end of SETUP, two additional states are required to ensure that memory is accessed properly. The current memory value is an input to the DBM (section 3.1 & 3.2) and the input to the instruction register (IR). During the FETCH_2 state the load line on the IR is asserted.

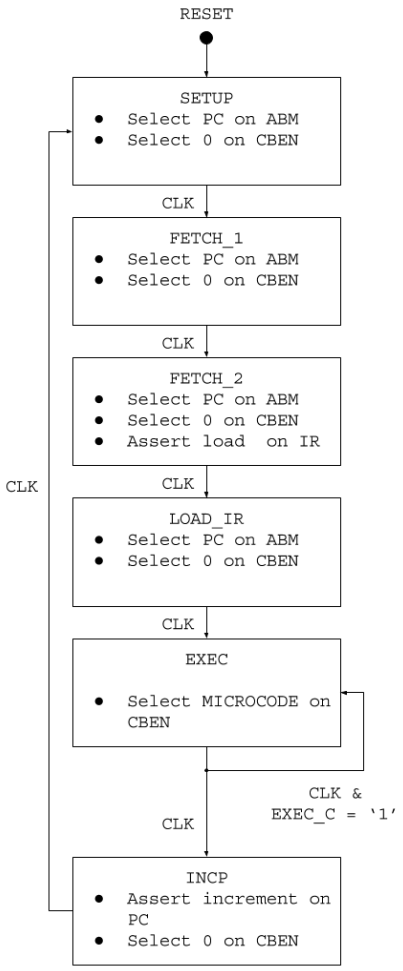

Figure 2: M2CPU Finite State Machine State Diagram

The fourth state LOAD_IR, loads the IR. During the LOAD_IR state the control bus is disabled and the program counter is applied to the address bus. This ensures that the FSM is still in control of the processor. At this point in the machine cycle the processor has loaded a new instruction and is ready to begin executing it.

On the next clock edge the control bus is enabled, thereby passing control to the microcode. This is the EXEC state. Each 'microinstruction' that is executed represents an instruction specific 'microstate' that the processor

enters while remaining in the EXEC 'macrostate'. Each microinstruction that is followed by another microinstruction must signal the FSM to ensure that it remains in the EXEC 'macrostate'. The last microinstruction in an instruction does not signal the FSM, thereby allowing it to continue to the final state in the machine cycle.

The sixth and final state in the M2CPU's machine cycle is INCPC state. In the INCPC state the control bus is set to 0, passing control back to the FSM, and the program counter is incremented. The processor is now ready to begin fetching the next instruction in the program.

4.2 Microcode

The M2CPU's microcode provides a means to translate an instruction into a control bus state. The M2CPU' microcode is stored in a look up table. The instruction held in the IR acts as an address and is fed into the look up table. The output of the look up table is applied to the control bus enable multiplexer (CBEN in figure 1). When in the EXEC state the CBEN is set to apply the microcode to the control bus. Every microinstruction that is followed by another microinstruction must signal the FSM to remain in the EXEC state. This means that the last microinstruction in a 'microprogram' (macroinstruction) must not signal the FSM, thereby allowing the machine cycle to continue. Additionally each microinstruction must load the next microinstruction in the 'microprogram'.

The contents of the IR act as a 'microaddress' for the 'microprogram' that implements an instruction. As such the IR's value must be changed in order to address a new microinstruction. It makes sense to store microinstructions in sequential microaddresses as this allows the next microinstruction in a microprogram to be indexed simply by incrementing the value of the IR. This is exactly how the M2CPU works. The IR includes logic to allow it to be incremented and thereby index the next microinstruction. Since the condition for signalling the FSM to stay in the EXEC state and the condition for incrementing the IR are the same a single bit of the control bus can be used to do both.

A downside of incrementing the IR to index a new microinstruction is that some instruction codes cannot be used. If a given instruction C requires three microstates to complete then the IR must be incremented twice after being loaded with the code representing C. This means that the values C, $C + 1$, and $C + 2$, are all addresses in the same microprogram. Therefore

no other instruction may be represented with the codes $C + 1$ or $C + 2$. When an macroinstruction is said to be represented by a given opcode C what is really meant is that the opcode C is the first microaddress in the microprogram that implements that macroinstruction. Table 1 shows which opcode each instruction in the M2 instruction set maps to. The shaded cells in table 1 are microaddresses which are only steps in microprograms. The labelled cells are the first addresses in their respective microprograms, i.e. 'the opcode' for that instruction.

5 VHDL Implementation

In general the VHDL implementation of the M2CPU closely follows the architecture diagram presented in section 3 (figure 1). Each block in figure 1 corresponds to a VHDL file in the processor's source tree. The dotted box in figure 1 corresponds to the `central_processing_unit.vhd` file. The other blocks correspond to similarly named files. The VHDL source differs from the diagram in a few areas. The bus multiplexers: ABM, DBM, CBEN, are not implemented as distinct entities but are instead part of the `central_processing_unit.vhd` file. The GPRs are multiple instances of the `general_purpose_register.vhd` entity rather than distinct entities. `general_purpose_register.vhd` and most SPRs are implemented as simple 8-bit registers with some additional logic. The underlying 8-bit registers are implemented as instances of the `register_8bit.vhd` entity. Some other important implementation details relating to bus architecture, branch instruction decoding, and the `JMP` instruction are explained in the following subsections.

The processor's source should be relatively easy to follow once one is familiar with figure 1 and the abbreviations used throughout this document. As such a detailed description of what each file does is largely unneeded. If one wishes for greater detail about the processor's implementation one is encouraged to read through the VHDL source.

5.1 Bus Architecture Implementation

Buses are often represented as strictly passive, i.e. wires. This presents a problem when multiple logical outputs are attached to a single bus: suppose one device wishes to assert a value on the bus, what value should the other devices on the bus assert? In discrete logic this problem is often resolved with tri-state logic. While tri-state logic can be modelled in VHDL it is awkward to do so. As such The M2CPU's implementation avoids this issue by implementing buses as multiplexers. For example; all devices that can assert data to the data bus are attached to the inputs of a data bus multiplexer (DBM). The output of the DBM is then attached to the input of every device that reads from the data bus.

5.2 Branch Instruction Implementation

The branch instructions (sections 2.4.2 and 6.11) conditionally increment the program counter based on the value of the processor's status flags. Since there are 8 branch instructions the instruction bus includes 8 individual 'branch-bits'. During a branch instruction the appropriate 'branch-bit' is set. If the condition associated with that branch-bit is met then the `PC` is incremented, otherwise it's value is left untouched. Determining whether or not the branch condition has been met is handled by the 'branch decoder' (`branch_decoder.vhd` in the source tree). The branch decoder AND's the branch-bits and the processor's status flags (or the inverse of the status flags). The resultant bus is then OR'ed together producing a single bit result. This single bit is applied to the `PC`'s increment line.

Since some other instructions also need access to the `PC`'s increment line The branch decoder also OR's these inputs with the single bit described above. This has an important implication for microcode, namely, it is possible to construct contradictory microinstructions. The processor has no safety mechanisms to deal with buggy microcode. It is up to the programmer to ensure that microinstructions are contradiction free.

5.3 Jump Instruction Implementation

At the end of every machine cycle the program counter is incremented (section 4.1). This presents a problem for the unconditional jump instruction (sections 2.4.1, and 6.13). The jump instruction is intended to set the address of the next instruction to be executed as the value stored in the `GH` register. Simply loading the program counter with the value stored in `GH` would cause the address of the next instruction to be `GH` + 1 since the program counter is incremented at the end of the machine cycle. To resolve this problem the `JMP` instruction loads the value `GH` - 1 into the program counter thereby correcting for the end of cycle increment. The subtraction operation is handled as part of the ABM. Essentially the value `GH` - 1 is added as a another input to the ABM. This is implemented in the file `central_processing_unit.vhd`.

6 Detailed Instruction Descriptions

This section provides a detailed description of each instruction in the M2 instruction set. Some instructions are very similar, as such they have been grouped for convenience. A description of each instruction's microstates is also included. This section is included primarily as technical reference. High level descriptions of the instruction set and the microcode implementation are provided in sections 2 and 4 respectively. Table 1 (section 6.14) provides a handy programming reference.

In this section *italics* denote variables. The values that a variable may take are defined in each subsection. `Monospace` denotes literal code. `UPPERCASE MONOSPACE` denotes M2 instruction mnemonics or M2 register names. `lowercase monospace` denotes port names in the VHDL code. For example `LDA X` is an instruction that the M2CPU can execute, whereas `la` is a port present on each GPR in the VHDL code.

6.1 Load GPR

These instructions load q with the value of r, where q and r are general purpose registers. Mnemonics take the form: LDq r.

- Mnemonics:
  - `LDA G`, `LDA H`, `LDA X`, `LDA Y`
  - `LDG A`, `LDG H`, `LDG X`, `LDG Y`
  - `LDH A`, `LDH G`, `LDH X`, `LDH Y`
  - `LDX A`, `LDX G`, `LDX H`, `LDX Y`
  - `LDY A`, `LDY G`, `LDY H`, `LDY X`
- States:
  - State 1: Select r on the data bus mux. Assert `la` on q.
  - State 2: Unassert `la` on q. Maintain DBM selection.

6.2 Load SPR

These instructions are conceptually similar to the load GPR instructions (section 6.1) but have varying microcode implementations due to the fact that each SPR is implemented and connected in a unique manner.

This instruction Loads the combined `GH` register with the value of the `PC`. The reverse operation; loading the `PC` with the value of the `GH` register is implemented by the 'jump' instruction (section 6.13).

- Mnemonic:
  - `LDGH PC`
- States:
  - State 1: Select `GH` on the address bus mux. Assert load lines (`lb` on `G` and `H` respectively) on the combined `GH` register.
  - State 2: Unassert load lines on `G`. Maintain ABM selection.

These instructions load the `X` register with the stack pointer or the stack pointer with the `X` register.

- Mnemonics:
  - `LDSP X`
  - `LDX SP`
- States:
  - State 1: Assert `lb` on the `X` register for `LDX SP`, or assert `ld` on `SP` for `LDSP X`.

Load the `Y` register with the value of the status register.

- Mnemonic:
  - `LDY S`
- States:
  - State 1: Assert `lb` on the `Y` register.

6.3 Load Memory From GPR

These instructions Load memory at the address held in the `GH` register with the value held in q. Where q is one of the GPRs.

- Mnemonics:
  - `LDM A`
  - `LDM G`
  - `LDM H`
  - `LDM X`
  - `LDM Y`
- States:
  - State 1: Select `GH` on the ABM. Select q on the DBM.
  - State 2: Assert `memory_wren`. Maintain bus multiplexer selections (memory access 1).
  - State 3: Unassert `memory_wren`. Maintain bus multiplexer selections.

6.4 Load GPR From Memory

Load q with the value held in memory at the address held in the `GH` register. Where q is a GPR.

- Mnemonics:
  - `LDA M`
  - `LDG M`
  - `LDH M`
  - `LDX M`
  - `LDY M`
- States:
  - State 1: Select `GH` on the ABM. Select memory on the DBM.
  - State 2: Maintain bus multiplexer selections (memory access 1)

  - State 3: Assert `la` on q. Maintain bus multiplexer selections (memory access 2).
  - State 4: Unassert `la`. Maintain bus multiplexer selections.

6.5 Load GPR With Literal

These instructions Load q with a literal hexadecimal value (represented by a #). Where q is a GPR. The literal is stored in the address directly following the instruction. This means that this instruction assembles to two addresses; one for the instruction code and one for the constant value.

- Mnemonics:
  - `LDA #`
  - `LDG #`
  - `LDH #`
  - `LDX #`
  - `LDY #`
- States:
  - State 1: Select the `PC` on the ABM. Select memory on the DBM. Assert `incpc`.
  - State 2: Unassert `incpc`. Maintain bus multiplexer selections (memory access 1).
  - State 3: Maintain bus multiplexer selections (memory access 2).
  - State 4: Assert `la` on q.
  - State 5: Unassert `la` on q.

6.6 Push GPR

these instructions push the contents of q to the zero page stack. Where q is a GPR. The stack grows from low addresses to high addresses so this operation increments the stack pointer.

- Mnemonics:

  - `PHA`
  - `PHG`
  - `PHH`
  - `PHX`
  - `PHY`
- States:
  - State 1: Select the `SP` on the ABM. Select q on the DBM.
  - State 2: Assert `memory_wren`. Maintain bus multiplexer selections (memory access 1).
  - State 3: Unassert `memory_wren`. Assert `ph` on `SP`.
  - State 4: Unassert `ph` on `SP`.

6.7 Pop GPR

These instructions pop the top of the zero page stack into q. Where q is a GPR. The stack grows from low addresses to high addresses so this operation decrements the stack pointer.

- Mnemonics:
  - `PPA`
  - `PPG`
  - `PPH`
  - `PPX`
  - `PPY`
- States:
  - State 1: Select `SP` on the ABM. Select memory on the DBM. Assert `pp` on `SP`.
  - State 2: Unassert `pp` on `SP`. Maintain bus multiplexer selections.
  - State 3: Maintain bus multiplexer selections (memory access 1).
  - State 4: Maintain bus multiplexer selections (memory access 2).

- State 5: Assert `la` on q.
- State 6: Unassert `la`.

6.8 ALU Operations

These instructions perform the selected operation between the `X` and `Y`(or just `X`) registers and store the result in the `A` register. Write status flags in the status register. Eight operations are supported, the description of each instruction follows its mnemonic.

- Mnemonics:
  - `ADD`: store `X` + `Y` in `A`
  - `SUB`: store `X` - `Y` in `A`
  - `AND`: store `X` AND `Y` in `A`
  - `NND`: store `X` NAND `Y` in `A`
  - `ORR`: store `X` OR `Y` in `A`
  - `XOR`: store `X` XOR `Y` in `A`
  - `STL`: shift `X` towards the left. LSb becomes 0. store result in `A`. Carry flag is set to old MSb.
  - `STR`: shift `X` towards the right. MSb becomes 0. store result in `A`. Carry flag is set to old LSb.
- States:
  - State 1: Select operation on ALU operation bus. Assert `lb` on `A`. Assert `ld` on status register.
  - State 2: Unassert `lb` and `ld`. Maintain ALU operation bus selection.

6.9 Set Status Flag

These instructions (S)et the (r) (F)lag. Where r is one of the processor's status flags: (Z)ero, (N)egative, (C)arry (O)overflow.

- Mnemonics :

  - `SZF`
  - `SNF`
  - `SCF`
  - `SOF`
- States:
  - State 1: Assert the appropriate flag set line on the status register.

6.10 Clear Status Flag

These instructions (C)lear the (r) (F)lag. Where r is one of the processor's status flags: (Z)ero, (N)egative, (C)arry (O)overflow.

- Mnemonics
  - `CZF`
  - `CNF`
  - `CCF`
  - `COF`
- States:
  - State 1: Assert the appropriate flag clear line on the status register.

6.11 Conditional Branch

These instructions cause program flow to branch if the specified condition is true. The condition evaluated is the state of one of the processor's status flags. The next instruction is executed if the condition is false. Otherwise skip the next instruction and execute the one that follows it instead. Note: a false result gets only one instruction to use before the true branch is executed. For more information on conditional branches see section 2.4.2. Mnemonics are of the form '(B)ranch if r is (S)et' or '(B)ranch if r is (C)lear'. Where r is one of the processor's status flags: (Z)ero, (N)egative, (C)arry (O)overflow.

- Mnemonics:
  - `BZS`

  - BZC
  - BNS
  - BNC
  - BCS
  - BCC
  - BOS
  - BOC

- States:
  - State 1: Assert appropriate branch line on the control bus. The branch lines are ANDed with the appropriate flag (or NOT flag) and the result is applied to `incpc` on the program counter.

6.12 No-operation

This instruction does nothing for one EXEC state. This instruction takes 6 clks to complete. For more information on the no-operation see section 2.4.

- Mnemonic:
  - NOP
- States:
  - State 1: Set control bus to 0.

6.13 Jump

This instruction sets the PC so that the next instruction executed is the one stored at the address held in the GH register (i.e. (J)u(MP) to GH). Note that since a machine cycle ends with the PC incrementing it is necessary to load the PC with the address held in GH less 1. For more information on the Jump instruction see section 2.4.1. For more information on the M2CPU's machine cycle see section 4.1.

- Mnemonic:
  - JMP

- States:
  - State 1: Select `GH - 1` on the ABM. Assert `ld` on the `PC`.
  - state 2: Unassert `ld` on the `PC`. Maintain bus multiplexer selections.

6.14 Instruction Table

Table 1 shows each instruction and its corresponding hexadecimal opcode. Each instruction is a single byte long, however some instructions take a single-byte literal argument represented with a '`#`'. The hexadecimal opcode (microaddress) for a a given instruction is found by adding the high nybble (row) and low nybble (column). Detailed descriptions each instruction are provided in sections 6.1 to 6.13. The shaded cells represent microaddresses that contain intermediary microstates needed to implement the listed opcodes. Since opcodes increase from left to right in table 1 the number of shaded cells following an opcode is a direct indication of how long that instruction takes to complete. For example: `PPA` takes 3 times as long to execute as `LDA X`. For more detail on the specifics of the machine cycle and instruction execution see section 4.

		LOW NYBBLE															
		0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
	0x00	NOP	LDA G		LDA H		LDA X		LDA Y		LDSP X			ADD		BZS	
	0x10		LDG A		LDG H		LDG X		LDG Y		LDX SP			SUB		BZC	
H	0x20		LDH A		LDH G		LDH X		LDH Y		LDY S			AND		BNS	
I	0x30		LDX A		LDX G		LDX H		LDX Y		LDGH PC			NND		BNC	
G	0x40		LDY A		LDY G		LDY H		LDY X		JMP			ORR		BCS	
H	0x50													XOR		BCC	
	0x60	LDM A			LDA M				LDA #					STL		BOS	
N	0x70	LDM G			LDG M				LDG #					STR		BOC	
Y	0x80	LDM H			LDH M				LDH #							SZF	
B	0x90	LDM X			LDX M				LDX #							CZF	
B	0xA0	LDM Y			LDY M				LDY #							SNF	
L	0xB0	PHA					PPA									CNF	
E	0xC0	PHG					PPG									SCF	
	0xD0	PHH					PPH									CCF	
	0xE0	PHX					PPX									SOF	
	0xF0	PHY					PPY									COF	

Table 1: The M2 Instruction Set