**Processor Design Project**

**Interim Report**

**EE480**

Team Members: Alex Hendren & Zachary Sean McFeely

3/25/2013

Table of Contents

[Table of Figures 4](#_Toc352703169)

[1. Introduction 5](#_Toc352703170)

[1.1. Implementation 5](#_Toc352703171)

[1.1.1. Add 5](#_Toc352703172)

[1.1.2. Sub 6](#_Toc352703173)

[1.1.3. Logical OR 6](#_Toc352703174)

[1.1.4. Logical AND 7](#_Toc352703175)

[1.1.5. Logical Complement (COMP) 8](#_Toc352703176)

[1.1.6. Multiply and Divide 8](#_Toc352703177)

[1.1.7. Arithmetic Left/Right Shift 9](#_Toc352703178)

[1.1.8. Conditional Branch 10](#_Toc352703179)

[1.1.9. Unconditional Jump 11](#_Toc352703180)

[1.1.10. Branch to a Subroutine 11](#_Toc352703181)

[1.1.11. Return from a Subroutine 12](#_Toc352703182)

[1.1.12. Return from Interrupt Service Routine 12](#_Toc352703183)

[1.1.13. LOAD Accumulator 13](#_Toc352703184)

[1.1.14. STORE Accumulator to RAM 14](#_Toc352703185)

[1.1.15. LOAD A Register from RAM 14](#_Toc352703186)

[1.1.16. STORE A Register to RAM 15](#_Toc352703187)

[1.1.17. LOAD B Register from RAM 16](#_Toc352703188)

[1.1.18. STORE B Register to RAM 17](#_Toc352703189)

[1.1.19. INPUT Data Word to RAM 17](#_Toc352703190)

[1.1.20. OUTPUT Data Word from RAM 18](#_Toc352703191)

[1.1.21. LOAD Mask Register of HVPI 19](#_Toc352703192)

[1.1.22. NOP 19](#_Toc352703193)

[2. Accumulator Architecture Development 20](#_Toc352703194)

[2.1. Architecture Overview 20](#_Toc352703195)

[2.2. Instruction Implementation 22](#_Toc352703196)

[2.2.1. Add 24](#_Toc352703197)

[2.2.2. Sub 25](#_Toc352703198)

[2.2.3. Logical OR 26](#_Toc352703199)

[2.2.4. Logical AND 27](#_Toc352703200)

[2.2.5. Logical Complement (COMP) 27](#_Toc352703201)

[2.2.6. Multiply and Divide 28](#_Toc352703202)

[2.2.7. Arithmetic Left/Right Shift 30](#_Toc352703203)

[2.2.8. Conditional Branch 31](#_Toc352703204)

[2.2.9. Unconditional Jump 31](#_Toc352703205)

[2.2.10. Branch to a Subroutine 32](#_Toc352703206)

[2.2.11. Return from a Subroutine 32](#_Toc352703207)

[2.2.12. LOAD Accumulator from RAM 33](#_Toc352703208)

[2.2.13. STORE Accumulator to RAM 35](#_Toc352703209)

[2.2.14. LOAD A Register from RAM 36](#_Toc352703210)

[2.2.15. STORE A Register to RAM 37](#_Toc352703211)

[2.2.16. LOAD B Register from RAM 38](#_Toc352703212)

[2.2.17. STORE B Register to RAM 39](#_Toc352703213)

[2.2.18. INPUT Data Word to RAM 40](#_Toc352703214)

[2.2.19. OUTPUT Data Word from RAM 41](#_Toc352703215)

[2.2.20. LOAD Mask Register of HVPI 42](#_Toc352703216)

[2.2.21. NOP 42](#_Toc352703217)

[2.3. Direct Mapped Cache 43](#_Toc352703218)

[3. Conclusion 43](#_Toc352703219)

[4. References 44](#_Toc352703220)

Table of Figures

[Figure 1 - ACC Processor Single Cycle Diagram 21](#_Toc352704788)

[Figure 2 - Start of Single Cycle Instruction Cycle 23](#_Toc352704789)

[Figure 3 - ADD Flow Diagram 24](#_Toc352704790)

[Figure 4 - SUB Flow Diagram 25](#_Toc352704791)

[Figure 5 - OR Flow Diagram 26](#_Toc352704792)

[Figure 6 - AND Flow Diagram 27](#_Toc352704793)

[Figure 7 - COMP Flow Diagram 27](#_Toc352704794)

[Figure 8 - MULT Flow Diagram 28](#_Toc352704795)

[Figure 9 - DIV Flow Diagram 29](#_Toc352704796)

[Figure 10 - SHFT Flow Diagram 30](#_Toc352704797)

[Figure 11 - BRA Flow Diagram 31](#_Toc352704798)

[Figure 12 - JMP Flow Diagram 31](#_Toc352704799)

[Figure 13 - BSR Flow Diagram 32](#_Toc352704800)

[Figure 14 - RTS Flow Diagram 32](#_Toc352704801)

[Figure 15 - LOAD Flow Diagram 34](#_Toc352704802)

[Figure 16 - STORE Flow Diagram 35](#_Toc352704803)

[Figure 17 - LDA Flow Diagram 36](#_Toc352704804)

[Figure 18 - STA Flow Diagram 37](#_Toc352704805)

[Figure 19 - LDB Flow Diagram 38](#_Toc352704806)

[Figure 20 - STB Flow Diagram 39](#_Toc352704807)

[Figure 21 - INPUT Flow Diagram 40](#_Toc352704808)

[Figure 22 - OUTPUT Flow Diagram 41](#_Toc352704809)

[Figure 23 - LMSK Flow Diagram 42](#_Toc352704810)

[Figure 24 - NOP Flow Diagram 42](#_Toc352704811)

[Figure 25 - Direct Mapped Cache Flow Diagram 43](#_Toc352704812)

# Introduction

The purpose of this project is to design, develop, test, and validate a high performing accumulator based processor capable of efficiently implementing an Instruction Set Architecture (ISA) that we specifically designed for it. By definition, the accumulator based architecture has only one register, called the accumulator, in the data path and for all ALU operations. For every ALU instruction, one operand will always be in the ACC register and the other operand comes from RAM. The process of developing the accumulator processor begins with defining the ISA and designing the physical layout of the architecture. After the architecture is designed, each clock cycle operation of the processor is defined as to show the processors operation and functionality as a system. From this, the steps of each instruction are diagramed on a cycle by cycle basis to define their exact operation by the system on the architecture.

# Instruction Set Architecture Development

The instruction set architecture defines the computer architecture and capabilities by detailing the data types, possible instructions, registers, memory addressing modes, interrupt control, and data input/output control. Defining the instruction set architecture enables programming the computer architecture, assembly instructions and associated parameters are translated to machine code.

## Implementation

This accumulator computer architecture includes 16 opcodes with multiple flags, which describe addressing modes and specific opcode operation. Each instruction is implemented as 16-bits wide (one word) for this accumulator processor, which features an 8-bit data bus. 5-bits are reserved for the opcode, 3-bits are reserved for the opcode flag, and 8-bits are allocated for the operand. Table 1 provides an overview of the instruction structure. The size of the operand is limited by the width of the data bus (5-bits).

Table 1 - Instruction Structure

|  |  |  |  |
| --- | --- | --- | --- |
| Bits | 15:11 | 10:8 | 7:0 |
| Field | OPCODE | FLAG | OPERAND |

Each of the 16 opcodes are detailed in the following sections by OPCode, flags, assembly format, machine code format, description of operation, architecture level operation, and memory addressing.

### Add

|  |  |
| --- | --- |
| Instruction | Add |
| OPCode | 00000 |
| Flags | 000: Direct 001: Indirect 010: Immediate |
| Format | ADD FLAG OPERAND |
| Machine Format | |  |  |  |  | | --- | --- | --- | --- | | Bits | 15:11 | 10:8 | 7:0 | | Field | OPCODE | FLAG | OPERAND | | Values | 00000 | xxx | RAM ADDR / INTEGER | |
| Description | Adds the specified operand to the current contents of the ACC register. The Output from the ALU will be returned to the ACC register. |
| Operation | ACC ← ACC + OPERAND |
| Memory Addressing | Immediate, Direct, and Indirect |

### Sub

|  |  |
| --- | --- |
| Instruction | SUB |
| OPCode | 00001 |
| Flags | 000: Direct 001: Indirect 010: Immediate |
| Format | SUB FLAG OPERAND |
| Machine Format | |  |  |  |  | | --- | --- | --- | --- | | Bits | 15:11 | 10:8 | 7:0 | | Field | OPCODE | FLAG | OPERAND | | Values | 00001 | xxx | RAM ADDR / INTEGER | |
| Description | Subtracts the specified operand from the current contents of the ACC register. The Output from the ALU will be returned to the ACC register. |
| Operation | ACC ← ACC - OPERAND |
| Memory Addressing | Immediate, Direct, and Indirect |

### Logical OR

|  |  |
| --- | --- |
| Instruction | OR |
| OPCode | 00011 |
| Flags | 000: Direct 001: Indirect 010: Immediate |
| Format | OR FLAG OPERAND |
| Machine Format | |  |  |  |  | | --- | --- | --- | --- | | Bits | 15:11 | 10:8 | 7:0 | | Field | OPCODE | FLAG | OPERAND | | Values | 00011 | xxx | RAM ADDR / INTEGER | |
| Description | Perform a LOGICAL OR operation of contents in Accumulator and OPERAND. The result of the operation will be stored in the accumulator. |
| Operation | ACC ← ACC | OPERAND |
| Memory Addressing | Immediate, Direct, and Indirect |

### Logical AND

|  |  |
| --- | --- |
| Instruction | AND |
| OPCode | 00100 |
| Flags | 000: Direct 001: Indirect 010: Immediate |
| Format | AND FLAG OPERAND |
| Machine Format | |  |  |  |  | | --- | --- | --- | --- | | Bits | 15:11 | 10:8 | 7:0 | | Field | OPCODE | FLAG | OPERAND | | Values | 00100 | xxx | RAM ADDR / INTEGER | |
| Description | Perform a LOGICAL AND operation of contents in Accumulator and OPERAND. The result of the operation will be stored in the accumulator. |
| Operation | ACC ← ACC & OPERAND |
| Memory Addressing | Immediate, Direct, and Indirect |

### Logical Complement (COMP)

|  |  |
| --- | --- |
| Instruction | COMP |
| OPCode | 10000 |
| Flags | 000 |
| Format | COMP FLAG OPERAND |
| Machine Format | |  |  |  |  | | --- | --- | --- | --- | | Bits | 15:11 | 10:8 | 7:0 | | Field | OPCODE | FLAG | OPERAND | | Values | 10000 | 000 | 00000000 | |
| Description | Perform LOGICAL complement/Negation on the contents of the accumulator register. The complement will be stored in the accumulator register. |
| Operation | ACC ← NOT ACC |
| Memory Addressing | N/A |

### Multiply and Divide

|  |  |
| --- | --- |
| Instruction | MUL |
| OPCode | 00010 |
| Flags | 000: Direct 001: Indirect |
| Format | MUL FLAG OPERAND [Lower 4 bits] |
| Machine Format | |  |  |  |  | | --- | --- | --- | --- | | Bits | 15:11 | 10:8 | 7:0 | | Field | OPCODE | FLAG | OPERAND | | Values | 00010 | 000 001 | RAM ADDR | |
| Description | Performs multiplication on lower 4-bits of Accumulator and lower 4-bits of the operand. |
| Operation | ACC ← ACC \* OPERAND |
| Memory Addressing | Direct and Indirect |

|  |  |
| --- | --- |
| Instruction | DIV |
| OPCode | 00010 |
| Flags | 010: Direct 011: Indirect |
| Format | DIV FLAG OPERAND [Lower 4 bits] |
| Machine Format | |  |  |  |  | | --- | --- | --- | --- | | Bits | 15:11 | 10:8 | 7:0 | | Field | OPCODE | FLAG | OPERAND | | Values | 00010 | 010 011 | RAM ADDR | |
| Description | Performs division on lower 4-bits of Accumulator and lower 4-bits of the operand. |
| Operation | ACC ← ACC / OPERAND |
| Memory Addressing | Direct and Indirect |

### Arithmetic Left/Right Shift

|  |  |
| --- | --- |
| Instruction | SHFT |
| OPCode | 00101 |
| Flags | 000: Left Shift in 0 001: Left Shift in 1 |
| Format | SHFT FLAG OPPERAND |
| Machine Format | |  |  |  |  | | --- | --- | --- | --- | | Bits | 15:11 | 10:8 | 7:0 | | Field | OPCODE | FLAG | OPERAND | | Values | 00101 | 000 001 | INTEGER < 7 | |
| Description | Performs a LEFT SHIFT operation on the contents of the accumulator register. The value of the bit shifted in is dependent upon the provided flag. Any number of bits less than 7. |
| Operation | ACC ← ACC [N-2:0 + (FLAG BIT)\*OPERAND] |
| Memory Addressing | N/A |

|  |  |
| --- | --- |
| Instruction | SHFT |
| OPCode | 00101 |
| Flags | 010: Right Shift in 0 011: Right Shift in 1 |
| Format | SHFT FLAG OPERAND |
| Machine Format | |  |  |  |  | | --- | --- | --- | --- | | Bits | 15:11 | 10:8 | 7:0 | | Field | OPCODE | FLAG | OPERAND | | Values | 00101 | 010 011 | INTEGER < 7 | |
| Description | Performs a RIGHT SHIFT operation on the contents of the accumulator register. The value of the bit shifted in is dependent upon the provided flag. Shift by any number of bits less than 7. |
| Operation | ACC ← ACC [OPERAND\*(FLAG BIT) + N-1:1] |
| Memory Addressing | N/A |

### Conditional Branch

|  |  |
| --- | --- |
| Instruction | BRA |
| OPCode | 00110 |
| Flags | 000: Branch if Equal 001: Branch if Not Equal |
| Format | BRA FLAG OPERAND |
| Machine Format | |  |  |  |  | | --- | --- | --- | --- | | Bits | 15:11 | 10:8 | 7:0 | | Field | OPCODE | FLAG | OPERAND | | Values | 00110 | 000 001 | MEM ADDR | |
| Description | Performs a comparison between the Accumulator and A register. If true, jump to instruction pointed to by OPERAND.  The comparison is performed on the ACC and A registers. |
| Operation | PC ←OPERAND |
| Memory Addressing | Direct |

### Unconditional Jump

|  |  |
| --- | --- |
| Instruction | JMP |
| OPCode | 00111 |
| Flags | NULL |
| Format | JUMP OPERAND |
| Machine Format | |  |  |  |  | | --- | --- | --- | --- | | Bits | 15:11 | 10:8 | 7:0 | | Field | OPCODE | FLAG | OPERAND | | Values | 00111 | 000 | MEM ADDR | |
| Description | Executes an unconditional branch. When encountered the instruction pointer is adjusted to the operand target memory address. |
| Operation | IP ←IP + OPERAND |
| Memory Addressing | N/A |

### Branch to a Subroutine

|  |  |
| --- | --- |
| Instruction | BSR |
| OPCode | 10101 |
| Flags | NULL |
| Format | BSR OPERAND |
| Machine Format | |  |  |  |  | | --- | --- | --- | --- | | Bits | 15:11 | 10:8 | 7:0 | | Field | OPCODE | FLAG | OPERAND | | Values | 10101 | 000 | MEM ADDR | |
| Description | Transfer program control to the address specified by the operand. Before loading the program counter with the operand, the current contents of the ACC and PC should be saved to their respective stacks. |
| Operation | ACC\_STACK[0] ← ACC  PC\_STACK[0] ← PC  PC ← OPERAND |
| Memory Addressing | NULL |

### Return from a Subroutine

|  |  |
| --- | --- |
| Instruction | RTS |
| OPCode | 01000 |
| Flags | NULL |
| Format | RTS |
| Machine Format | |  |  |  |  | | --- | --- | --- | --- | | Bits | 15:11 | 10:8 | 7:0 | | Field | OPCODE | FLAG | OPERAND | | Values | 01000 | 000 | 00000000 | |
| Description | Transfer program control to the address located in the implied return address. Return is made to the top element in the PC stack. Should also pop the ACC value from the ACC stack. |
| Operation | PC ← PC\_STACK[0] ACC ← ACC\_STACK[0] |
| Memory Addressing | NULL |

### Return from Interrupt Service Routine

|  |  |
| --- | --- |
| Instruction | RTI |
| OPCode | 01001 |
| Flags | NULL |
| Format | RTI |
| Machine Format | |  |  |  |  | | --- | --- | --- | --- | | Bits | 15:11 | 10:8 | 7:0 | | Field | OPCODE | FLAG | OPERAND | | Values | 01001 | 000 | 00000000 | |
| Description | Return from an Interrupt Subroutine to the position the PC was at before the interrupt. The PC location to return to will be the top element in the PC stack. |
| Operation | PC ← Return Address (From Stack) |
| Memory Addressing | NULL |

### LOAD Accumulator

|  |  |
| --- | --- |
| Instruction | LOAD |
| OPCode | 01010 |
| Flags | 000: Direct 001: Indirect  010: Immediate |
| Format | LOAD FLAG OPERAND |
| Format Machine | |  |  |  |  | | --- | --- | --- | --- | | Bits | 15:11 | 10:8 | 7:0 | | Field | OPCODE | FLAG | OPERAND | | Values | 01010 | 000 001 010 | MEM ADDR MEM ADDR Integer | |
| Description | **Direct Mode** Copy the contents of the address in RAM specified by the source operand to the accumulator.  **Indirect Mode:** Copy the accumulator to the destination memory address found at the memory address of the operand  **Immediate Mode:** Copy the 8-bit immediate source operand to the accumulator. |
| Operation | **Direct Mode** ACC ← memory (OPERAND)  **Indirect Mode:**  ACC ← Memory{ memory( OPERAND) }   **Immediate Mode:** ACC ← IMMED OPERAND |
| Memory Addressing | Direct, Immediate |

### STORE Accumulator to RAM

|  |  |
| --- | --- |
| Instruction | STOR |
| OPCode | 01011 |
| Flags | 000: Direct 001: Indirect |
| Format | STORE FLAG OPERAND |
| Format Machine | |  |  |  |  | | --- | --- | --- | --- | | Bits | 15:11 | 10:8 | 7:0 | | Field | OPCODE | FLAG | OPERAND | | Values | 01011 | 000 001 | MEM ADDR | |
| Description | **Direct Mode** Copy the accumulator to the destination memory address specified by the operand **Indirect Mode:** Copy the accumulator to the destination memory address found at the memory address of the operand |
| Operation | **Direct Mode** memory(OPERAND) ← ACC **Indirect Mode:** Memory{ memory( OPERAND) } ← ACC |
| Memory Addressing | Direct, Indirect |

### LOAD A Register from RAM

|  |  |
| --- | --- |
| Instruction | LDA |
| OPCode | 10001 |
| Flags | 000: Direct 001: Indirect  010: Immediate |
| Format | LOAD FLAG OPERAND |
| Format Machine | |  |  |  |  | | --- | --- | --- | --- | | Bits | 15:11 | 10:8 | 7:0 | | Field | OPCODE | FLAG | OPERAND | | Values | 10001 | 000 001 010 | MEM ADDR MEM ADDR Integer | |
| Description | **Direct Mode** Copy the contents of the address in RAM specified by the source operand to the A register.  **Indirect Mode:** Copy the memory data found at the memory address of the operand into the A register.   **Immediate Mode:** Copy the 8-bit immediate source operand to the A register. |
| Operation | **Direct Mode** A ← memory (OPERAND)  **Indirect Mode:**  A ← Memory{ memory( OPERAND) }   **Immediate Mode:** A ← IMMED OPERAND |
| Memory Addressing | Direct, Immediate |

### STORE A Register to RAM

|  |  |
| --- | --- |
| Instruction | STA |
| OPCode | 10010 |
| Flags | 000: Direct 001: Indirect |
| Format | STORE FLAG OPERAND |
| Format Machine | |  |  |  |  | | --- | --- | --- | --- | | Bits | 15:11 | 10:8 | 7:0 | | Field | OPCODE | FLAG | OPERAND | | Values | 10010 | 000 001 | MEM ADDR | |
| Description | **Direct Mode** Copy the A register contents to the destination memory address specified by the operand **Indirect Mode:** Copy the A register contents to the destination memory address found at the memory address of the operand |
| Operation | **Direct Mode** memory(OPERAND) ← A **Indirect Mode:** Memory{ memory( OPERAND) } ← A |
| Memory Addressing | Direct, Indirect |

### LOAD B Register from RAM

|  |  |
| --- | --- |
| Instruction | LDB |
| OPCode | 10011 |
| Flags | 000: Direct 001: Indirect  010: Immediate |
| Format | LOAD FLAG OPERAND |
| Format Machine | |  |  |  |  | | --- | --- | --- | --- | | Bits | 15:11 | 10:8 | 7:0 | | Field | OPCODE | FLAG | OPERAND | | Values | 10011 | 000 001 010 | MEM ADDR MEM ADDR Integer | |
| Description | **Direct Mode** Copy the contents of the address in RAM specified by the source operand to the B register.  **Indirect Mode:** Copy the memory data found at the memory address of the operand into the B register.   **Immediate Mode:** Copy the 8-bit immediate source operand to the B register. |
| Operation | **Direct Mode** B ← memory (OPERAND)  **Indirect Mode:**  B ← Memory{ memory( OPERAND) }   **Immediate Mode:** B ← IMMED OPERAND |
| Memory Addressing | Direct, Immediate |

### STORE B Register to RAM

|  |  |
| --- | --- |
| Instruction | STB |
| OPCode | 10100 |
| Flags | 000: Direct 001: Indirect |
| Format | STORE FLAG OPERAND |
| Format Machine | |  |  |  |  | | --- | --- | --- | --- | | Bits | 15:11 | 10:8 | 7:0 | | Field | OPCODE | FLAG | OPERAND | | Values | 10100 | 000 001 | MEM ADDR | |
| Description | **Direct Mode** Copy the B register contents to the destination memory address specified by the operand **Indirect Mode:** Copy the B register contents to the destination memory address found at the memory address of the operand |
| Operation | **Direct Mode** memory(OPERAND) ← B **Indirect Mode:** Memory{ memory( OPERAND) } ← B |
| Memory Addressing | Direct, Indirect |

### INPUT Data Word to RAM

|  |  |
| --- | --- |
| Instruction | INPUT |
| OPCode | 01100 |
| Flags | 000: Direct 001: Indirect |
| Format | INPUT FLAG OPERAND |
| Format Machine | |  |  |  |  | | --- | --- | --- | --- | | Bits | 15:11 | 10:8 | 7:0 | | Field | OPCODE | FLAG | OPERAND | | Values | 01100 | 000 001 | MEM ADDR | |
| Description | Input a data word to RAM |
| Operation | **Direct Mode** memory(OPERAND) ← I/O Port  **Indirect Mode:** Memory{ memory( OPERAND) } ← I/O Port  **Flow Control**   * Wait until processor is ready to receive INPUT data. * Wait for INPUT data to become ready from connected device. * Read in INPUT data. * Notify that INPUT data has been read. |
| Memory Addressing | Direct, Indirect |

### OUTPUT Data Word from RAM

|  |  |
| --- | --- |
| Instruction | OUTPUT |
| OPCode | 01101 |
| Flags | 000: Direct 001: Indirect |
| Format | OUTPUT FLAG OPERAND |
| Format Machine | |  |  |  |  | | --- | --- | --- | --- | | Bits | 15:11 | 10:8 | 7:0 | | Field | OPCODE | FLAG | OPERAND | | Values | 01101 | 000 001 | MEM ADDR | |
| Description | Output data word from RAM |
| Operation | **Direct Mode** memory(OPERAND) → I/O Port  **Indirect Mode:** Memory{ memory( OPERAND) } → I/O Port  **Flow Control**   * Wait until processor data is ready to OUTPUT. * Wait until connected device is ready to receive. * Output OUTPUT data. * Output data until connected device says the data has been received. |
| Memory Addressing | Direct, Indirect |

### LOAD Mask Register of HVPI

|  |  |  |
| --- | --- | --- |
| Instruction | LMSK | |
| OPCode | 01110 | |
| Flags | | NULL |
| Format | LMSK VALUE | |
| Format Machine | |  |  |  |  | | --- | --- | --- | --- | | Bits | 15:11 | 10:8 | 7:0 | | Field | OPCODE | FLAG | OPERAND | | Values | 01110 | 00 | 00001000 00000100 00000010 00000001 00000000 | | |
| Description | Loads the mask register of the Maskable Hardware Vectorized Priority Interrupt System. The lower 4 bits of the operand will load into the 4 bit mask register within the MHVPIS.  Value: 1 – 1000 – ALU Output Zero Value: 2 – 0100 – ALU Overflow Value: 3 – 0010 – Illegal Opcode Value: 4 – 0001 – INPUT/OUTPUT Interrupt Value: 0 – 0000 – CLEAR mask register (disables interrupts) | |
| Operation | MASK REGISTER ← VALUE[3:0] | |
| Memory  Addressing | NULL | |

### NOP

|  |  |
| --- | --- |
| Instruction | NOP |
| OPCode | 01111 |
| Flags | NULL |
| Format | NOP |
| Format Machine | |  |  |  |  | | --- | --- | --- | --- | | Bits | 15:11 | 10:8 | 7:0 | | Field | OPCODE | FLAG | OPERAND | | Values | 01111 | 000 | 00000000 | |
| Description | Performs no operation. |
| Operation | NULL |
| Memory Addressing | NULL |

# Accumulator Architecture Development

## Architecture Overview

Figure 1 provides a single cycle diagram highlighting the accumulator processor. The hardware architecture has been designed to support the operations available in the instruction set architecture. Each of the data paths (wires) in Figure 1 are 8-bits wide. Control point and select lines are not drawn out in Figure 1 in interest of preserving space. Two memories (RAM) are used to store program data and the instruction data (programs to be executed).

Multiplexers are used to select what data exists on varying data busses at a given time. It is not desirable to have multiple data trying to output onto one bus at the same time, the current value would be unknown. Each of the multiplexer’s select lines are attached to the controller via control point lines (CP). Depending upon the opcode interpreted by the controller, the control points are set to enable the correct data to be passed to the desired function units.

It is preferred for each instruction to have a minimal cycle time, enabling the processor to execute each instruction more quickly. Every use of the ALU function unit leaves the new result stored in the accumulator register, ACC. Operations to use, load, and store the ACC register contents should be easily accessible, meaning available in very few clock cycles. The layout of the single cycle processor enables very short paths to the next functional unit. Memory access is an expensive operation, to simulate this – memory operations loop 8 clock cycles before returning results. To avoid constantly running realizing the burden of memory accesses, a direct-mapped cache is implemented. The 4 word cache mimics the L1 or L2 cache on a modern system. If the target memory address exists in the cache, the data can be returned without experiencing the 8 clock cycle delay.

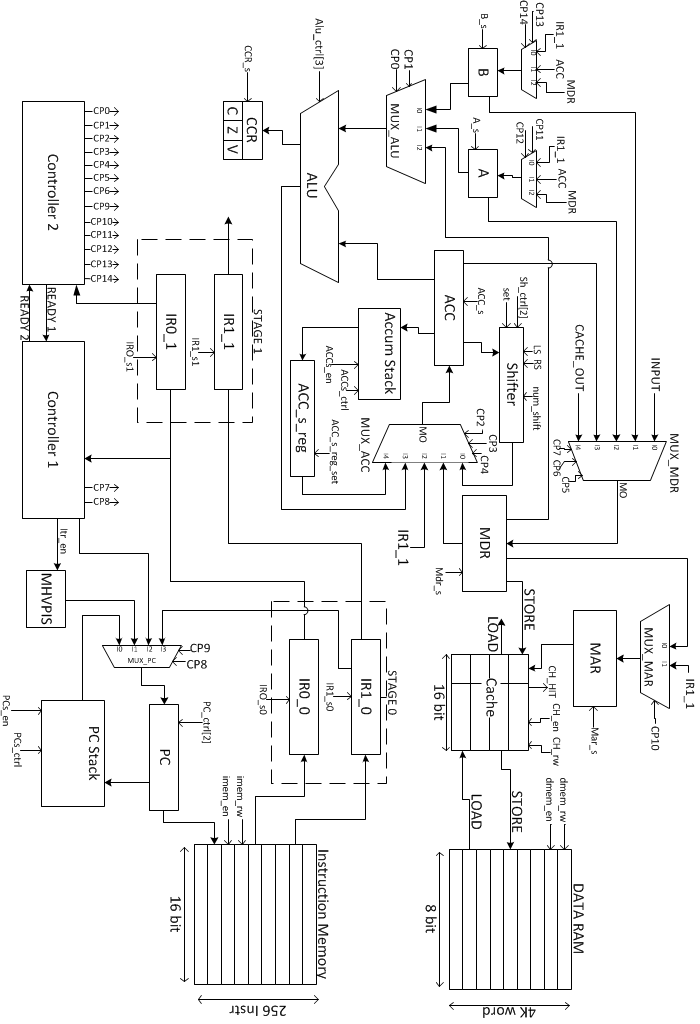


Figure 1 - ACC Processor Single Cycle Diagram

One ALU exists in the single cycle accumulator processor design. ALU operations must always include the ACC register as an input, and results are stored back into the ACC register. Operations enabling two inputs (Add, Subtract, etc.) allow input from the A, B, and MDR registers. A, B, and ACC may be loaded three different ways – direct memory addressing, indirect memory addressing, and using immediate addressing. A shifter is connected to the ACC register and enables the CPU to be programmed to shift the contents of the ACC register to the left or right by bits. Current contents of the A, B, and ACC registers may be saved into the processor’s data RAM using direct and indirect addressing schemes.

Interrupts are emitted and handled using a Masked Hardware Vectored Priority Interrupt System (MHVPIS). Divide by zero, overflow, input/output, and illegal Op-code instructions are all occurrences handled by the MHVPIS. When an interrupt is encountered, the MHVPIS points the program counter to the memory address for the interrupt service routine (ISR) to handle the interrupt.

Entering subroutines and interrupt service routines requires that the current value of the ACC register and PC be saved. Two stacks are introduced PC Stack and Accumulator Stack, to handle pushing the current register contents before entering the subroutine. When an instruction to return from subroutine is encountered the top of the stack is popped off and placed into the respective register.

## Instruction Implementation

Figure 2 highlights the initial operations performed at the beginning of every instruction execution. When control returns to the beginning of the loop, the controller must determine what the next instruction is highlighting by evaluating the Opcode and flags. Before the next instruction is read from the instruction memory, the MHVIPS is evaluated to check if an interrupt is pending. If there is an interrupt to be processed, the current contents of the PC and ACC are stored on their respective stacks and the interrupt service routine address is loaded into the program counter. Otherwise, the OPcode and instruction flag(s) into IR0 and the operand is loaded into IR1. Finally the PC is incremented to point to the next instruction in the instruction memory.



Figure 2 - Start of Single Cycle Instruction Cycle

After the initial operations have been completed, the instruction operations can be executed. The operation to be executed is determined within the controller by evaluating the contents of IR0 (OPcode + Flags). The following sections detail the processor cycles entailed for implementing each instruction. It is most desirable to have instructions complete using the minimal number of clock cycles to satisfy the instruction operation.

After each of the operations finish their loop, flow of control is directed to the pipeline handshake state. Figure 3 illustrates the logic state where the pipeline stage 1 handshaking occurs. In this state the controller communicates with the stage 0 controller to notify it that it is ready to receive an instruction. Stage 0 will notify stage 1 when it has an instruction ready to be executed. Once stage 1 begins executing the instruction, stage 1 notifies stage 0 that it has received the instruction and operand. This notification allows stage 0 to continue to its next cycle and begin fetching the next instruction. Stage 0 should generally finish always before stage 1, thus minimizing the amount of time stage 1 should have to ever wait for its next instruction to execute.



Figure 3 - Stage 1 Pipeline Handshake

### Add



Figure 4 - ADD Flow Diagram

### Sub



Figure 5 - SUB Flow Diagram

### Logical OR



Figure 6 - OR Flow Diagram

### Logical AND



Figure 7 - AND Flow Diagram

### Logical Complement (COMP)



Figure 8 - COMP Flow Diagram

### Multiply and Divide



Figure 9 - MULT Flow Diagram



Figure 10 - DIV Flow Diagram

### Arithmetic Left/Right Shift



Figure 11 - SHFT Flow Diagram

### Conditional Branch



Figure 12 - BRA Flow Diagram

### Unconditional Jump



Figure 13 - JMP Flow Diagram

### Branch to a Subroutine



Figure 14 - BSR Flow Diagram

### Return from a Subroutine



Figure 15 - RTS Flow Diagram

### Return from Interrupt Service Routine



Figure 16 - RTI State Flow

### LOAD Accumulator from RAM



Figure 17 - LOAD Flow Diagram

### STORE Accumulator to RAM



Figure 18 - STORE Flow Diagram

### LOAD A Register from RAM



Figure 19 - LDA Flow Diagram

### STORE A Register to RAM



Figure 20 - STA Flow Diagram

### LOAD B Register from RAM



Figure 21 - LDB Flow Diagram

### STORE B Register to RAM



Figure 22 - STB Flow Diagram

### INPUT Data Word to RAM



Figure 23 - INPUT Flow Diagram

### OUTPUT Data Word from RAM



Figure 24 - OUTPUT Flow Diagram

### LOAD Mask Register of HVPI



Figure 25 - LMSK Flow Diagram

### NOP



Figure 26 - NOP Flow Diagram

## Fully Associative Cache



Figure 27 – Fully Associative Cache Flow Diagram

# Conclusion

The current accumulator processor design is constructed using a single cycle model. To maximize performance, the controller will be built to support a two-stage pipeline. A two-stage pipeline will enable the processor to perform multiple operations under the same clock cycle; this is achieved by connecting components on individual bus lines thus preventing collision. Collision would occur when multiple functional units attempt to write to a bus during the same clock cycle. In addition to the pipelined approach, the processor implements a 4 word cache on the data memory thus improving latency associated with memory access. An MVHIPS is implemented to service interrupts as they occur during program execution by the processor. All of the components encompassed within the processor can be found in modern day processors, the problems to be faced will educate the team in dealing with real world design and troubleshooting exercises.

# References

To be updated as references are used.