MMARC-ONE©

(michael matthew architecture with one register)

Michael Baumgarten
Matthew Toro

MMARC-ONE©

Overview:

MMARC-ONE is an accumulator type architecture. It uses one register - the Accumulator - and memory to perform all operations. Every instruction is 16-bits long. 3-bits for the opcode and 13-bits for addresses or immediates. 13-bit addressing provides 2^{13} available bytes of memory. MMARC-ONE implements eight instructions, STOR, ADDA, ADDI, SUBA, LAAD, BIZI, MOVI, and SAAD.

Instruction Set:

- **STOR** *address* - **MEM[*address*] ← *Accumulator***
  - Store the accumulator to address
- **ADDA** *address* - ***Accumulator* ← MEM[*address*] + *Accumulator***
  - Add the value at a memory address to the accumulator
- **ADDI** *immediate* - ***Accumulator* ← *Accumulator* + *Immediate***
  - Add an immediate value to the accumulator
- **SUBA** *address* - ***Accumulator* ← *Accumulator* - MEM[*address*]**
  - Subtract a value at a memory address from the accumulator
- **LAAD** immediate - ***Accumulator* ← MEM[*Accumulator + immediate*]**
  - Load value at the address in the accumulator + an offset
- **BIZI** *label* - ***if (Accumulator = 0) then (PC ← PC + label offset)***
  - Branch if accumulator is zero
- **MOVI** *immediate* - ***Accumulator* ← *immediate***
  - Set the accumulator register to an immediate value
- **SAAD** *address* - **MEM[*Accumulator*] ← MEM[*address*]**
  - Set the value at the accumulator address to the value in memory at the parameter *address*

Instruction format:

OP CODE (3 bits)	ADDRESS/IMMEDIATE (13 bits)

Registers:

Accumulator - sole register available for math operations	(16 bits)
PC - program counter	(13 bits)

MMARC-ONE©

Memory:

A memory address is 13 bits long. This allows our architecture to access 2^{13} (8192) bytes of memory. A word is 2-bytes wide. Each instruction is also 2-bytes wide. Since instructions are of a fixed length, the PC is always updated by 2 to find the next instruction (PC ← PC + 2).

Immediates:

All immediates are 13 bits long. They are all signed values. Immediates can range from -4096 to 4095.

Encoding:

The MMARC-ONE requires 11 control signals which are determined by each instruction's op-code. These signals determine the flow of the datapath. Each instruction's control signal assignments are displayed in table one.

Table One - Control Signal Assignments

	OP CODE	branch	mem sel	read en	write en	accum write	alu op	alu src	in sel	imm sel	offset en	write sel
BIZI	"000"	1	x	0	0	0	x	x	x	x	x	x
SAAD	"001"	0	x	1	1	0	x	x	1	x	0	1
STOR	"010"	0	x	0	1	0	x	x	0	x	0	0
SUBA	"011"	0	0	1	0	1	1	1	x	0	0	x
ADDI	"100"	0	0	0	0	1	0	0	x	0	x	x
ADDA	"101"	0	0	1	0	1	0	1	x	0	0	x
MOVI	"110"	0	x	0	0	1	x	x	x	1	x	x
LAAD	"111"	0	1	1	0	1	0	0	x	0	1	x

MMARC-ONE©

Hardware:

MMARC-ONE's datapath was developed to cater to a single-register, memory-dependant architecture. The schematic is displayed in figure one.

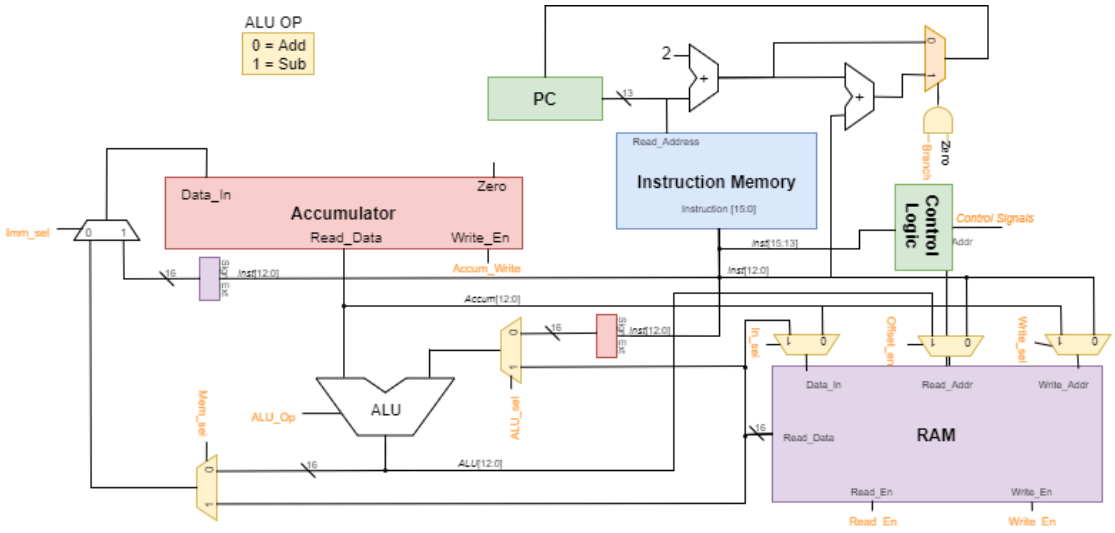

Figure One - MMARC-ONE Data Path

Each instruction contains an opcode which is used to configure the control signals. A PLA is configured to generate these logical assignments. The PLA used in the MMARC-ONE architecture is displayed in figure two.

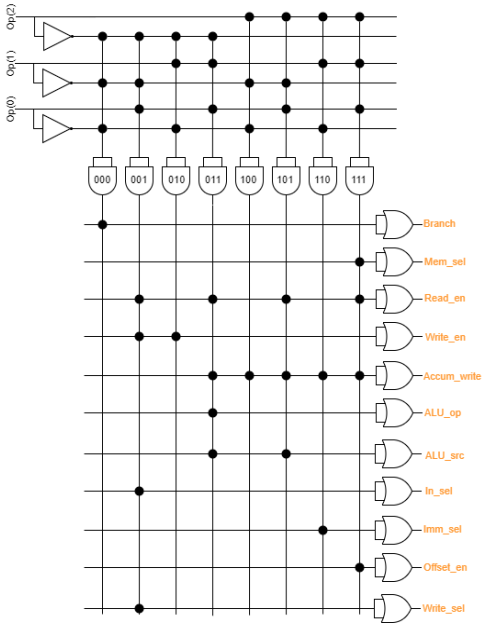

Figure Two - PLA

Timing:

To determine the estimated clock period, we assume that certain logic blocks in the architecture have the delays listed in table two.

Table Two - Execution Times

Logic Block	Time to Execute
Register	*1ns*
ALU	*2ns*
RAM	*2ns*
Instruction Memory	*2ns*

These values are used along with the critical path (longest instruction) to calculate the clock period. The critical path can be seen in Figure Three.

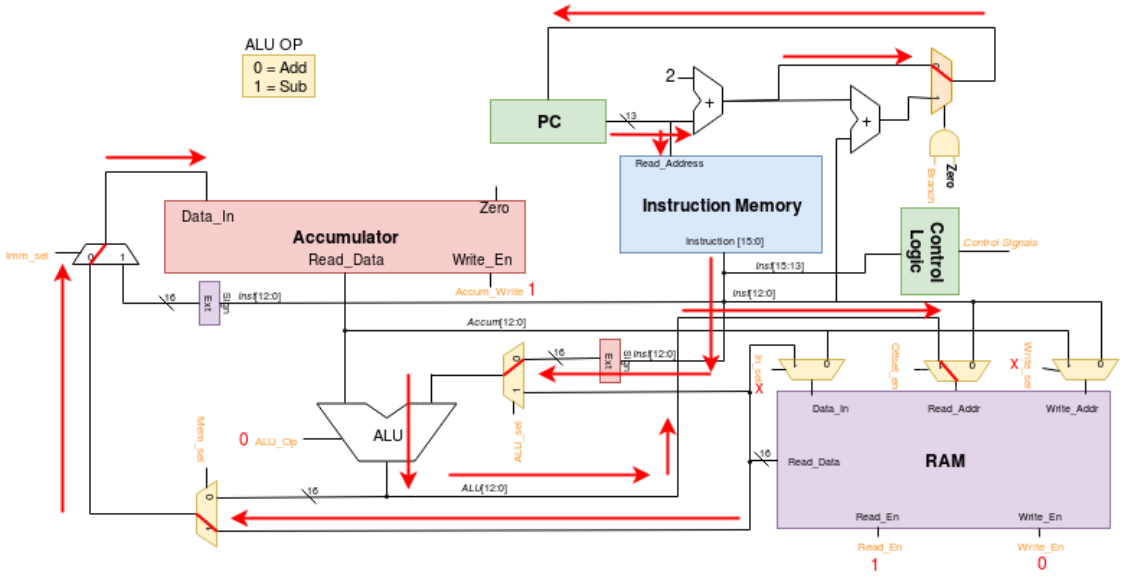

Figure Three - LAAD Instruction path

The critical path traverses four major logic blocks. The total execution time for this process is 2ns(InstMem) + 1ns(ALU) + 2ns(RAM) + 2ns(Accumulator) = 7ns. This instruction has the longest execution time. Therefore the CPU clock period should be at least 7ns.

Assembly Example Program:

AddTwoArrays adds two n-element arrays and stores them into a third sum array. It expects five predefined equates.

```
#def num_elements 0x0   ; points to a value containing the number of elements
#def arr_x_ptr    0x100 ; points to a pointer at the head of array x
#def arr_y_ptr    0x200 ; points to a pointer at the head of array y
#def arr_sum_ptr  0x300 ; points to a pointer at the head of array sum
#def temp         0x400 ; a temporary variable used in computation

       ; if zero elements left, end loop
Loop   LOAD   num_elements   ; accumulator       ← mem[num_elements]
       BIZ    EndLoop        ; while(num_elements !=0)
       ; load value from x_ptr and store at temp
       LOAD   arr_x_ptr      ; accumulator       ← mem[arr_x_ptr]
       LAAD   0              ; accumulator       ← mem[accumulator]
       STOR   temp           ; mem[temp]         ← accumulator
       ; load value from y_ptr and add temp to it
       LOAD   arr_y_ptr      ; accumulator       ← mem[arr_y_ptr]
       LAAD   0              ; accumulator       ← mem[accumulator]
       ADDA   temp           ; accumulator       ← accumulator + mem[temp]
       ; store result into temp and add to sum array
       STOR   temp           ; mem[temp]         ← accumulator
       LOAD   arr_sum_ptr    ; accumulator       ← mem[arr_sum_ptr]
       SAAD   temp           ; mem[accumulator]  ← mem[temp]
       ; increment x_ptr by 2
       LOAD   arr_x_ptr      ; accumulator       ← mem[arr_x_ptr]
       ADDI   #2             ; accumulator       ← accumulator + 2
       STOR   arr_x_ptr      ; mem[arr_x_ptr]    ← accumulator
       ; increment y_ptr by 2
       LOAD   arr_y_ptr      ; accumulator       ← mem[arr_y_ptr]
       ADDI   #2             ; accumulator       ← accumulator + 2
       STOR   arr_y_ptr      ; mem[arr_y_ptr]    ← accumulator
       ; increment sum_ptr by 2
       LOAD   arr_sum_ptr    ; accumulator       ← mem[arr_sum_ptr]
       ADDI   #2             ; accumulator       ← accumulator + 2
       STOR   arr_sum_ptr    ; mem[arr_sum_ptr]  ← accumulator
       ; decrement num elements by 1
       LOAD   num_elements   ; accumulator       ← mem[num_elements]
       ADDI   #-1            ; accumulator       ← accumulator - 1
       STOR   num_elements   ; mem[num_elements] ← accumulator
       ; loop back
       MOVI   #0             ; accumulator       ← #0
       BIZ    loop           ; PC                  ← loop
EndLoop
```