MMARC-12©

(mmarc one version 2)

Michael Baumgarten

Version 2.0 (Kumquat)

MMARC-12©

Overview:

MMARC-12 is an accumulator type architecture adapted from the MMARC-ONE. The design is based around a single register - the Accumulator - and memory to perform all operations. Each instruction is 24-bits long. 4-bits are used for the opcode and 20-bits for addresses or immediates. 20-bit addressing provides 2^{20} available bytes of memory. MMARC-12 implements sixteen instructions which permit complex software implementations.

Main Features:

- 1048576 accessible word addresses (3145728 bytes - 3.14 MB)
- Two alu operation flags for conditional branching
- Bitwise alu operations

Instruction format:

OP CODE (4 bits)	ADDRESS/IMMEDIATE (20 bits)

Registers:

Accumulator	- sole register available for math operations	(24 bits)
PC	- program counter	(20 bits)
Flags	- holds status bit flags (Zero and Negative)	(2 bits)

Memory:

A memory address is 20 bits long. This allows our architecture to access 2^{20} (1048576) words in memory. A word is 3-bytes wide. Each instruction is also 3-bytes wide. Each cycle the PC is always updated by 1 to find the next instruction (PC ← PC + 1).

Immediates:

All immediates are 20 bits long. They are all signed values. Immediates can range from -524288 to 524287.

MMARC-12©

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
- **SUBI** *immediate* - ***Accumulator* ← *Accumulator* - *Immediate***
  - Subtract an immediate value from the accumulator
- **LAAD** immediate - ***Accumulator* ← MEM[*Accumulator* + *immediate*]**
  - Load value at the address in the accumulator + an offset
- **BIZI** *label* - ***if (FLAGS[Z] = 0) then (PC ← PC + label offset)***
  - Branch if accumulator is zero
- **BRAM** *address* - ***PC ← MEM[address]***
  - Branch to the instruction at the given address
- **BILT** *label* - ***if (FLAGS[N] = 1) then (PC ← PC + label offset)***
  - Branch if the negative flag is set
- **BIGR** *label* - ***if (FLAGS[N] = 0 & FLAGS[Z] = 0) then (PC ← PC + label offset)***
  - Branch if the negative and zero flag are both cleared
- **CMPR** immediate - ***FLAGS ← RESULT[accumulator - immediate]***
  - Assign flag register bits with subtract operation
- **MOVI** *immediate* - ***Accumulator* ← *immediate***
  - Set the accumulator register to an immediate value
- **BAND** immediate - ***Accumulator* ← *Accumulator* && *immediate***
  - Bitwise AND operation on accumulator
- **BIOR** immediate - ***Accumulator* ← *Accumulator* || *immediate***
  - Bitwise OR operation on accumulator
- **BXOR** immediate - ***Accumulator* ← *Accumulator* ⊕ *immediate***
  - Bitwise XOR operation on accumulator
- **SAAD** *address* - **MEM[*Accumulator*] ← MEM[*address*]**
  - Set the value at the accumulator address to the value in memory at the parameter *address*

MMARC-12©

Encoding:

The MMARC-12 requires 10 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

Instruction	b3	b2	b1	b0	branch	branch type	mem select	write enable	flags write	accum write	alu src	alu op	mem in sel	accum in sel
STOR	0	0	0	0	0	x	x	1	0	0	x	x	0	x
ADDA	0	0	0	1	0	x	0	0	0	1	1	"000"	x	0
ADDI	0	0	1	0	0	x	0	0	0	1	0	"000"	x	0
SUBA	0	0	1	1	0	x	0	0	0	1	1	"001"	x	0
SUBI	0	1	0	0	0	x	0	0	0	1	0	"001"	x	0
BAND	0	1	0	1	0	x	0	0	0	1	0	100	x	0
BIOR	0	1	1	0	0	x	0	0	0	1	0	"010"	x	0
BXOR	0	1	1	1	0	x	0	0	0	1	0	"011"	x	0
LAAD	1	0	0	0	0	x	1	0	0	1	0	"000"	x	0
CMPR	1	0	0	1	0	x	x	0	1	0	0	"001"	x	x
MOVI	1	0	1	0	0	x	x	0	0	1	x	x	x	1
SAAD	1	0	1	1	0	x	x	1	0	0	x	x	1	x
BIZI	1	1	0	0	1	0	x	0	0	0	x	x	x	x
BRAM	1	1	0	1	1	1	x	0	0	0	x	x	x	x
BILT	1	1	1	0	1	0	x	0	0	0	x	x	x	x
BIGR	1	1	1	1	1	0	x	0	0	0	x	x	x	x

Hardware:

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

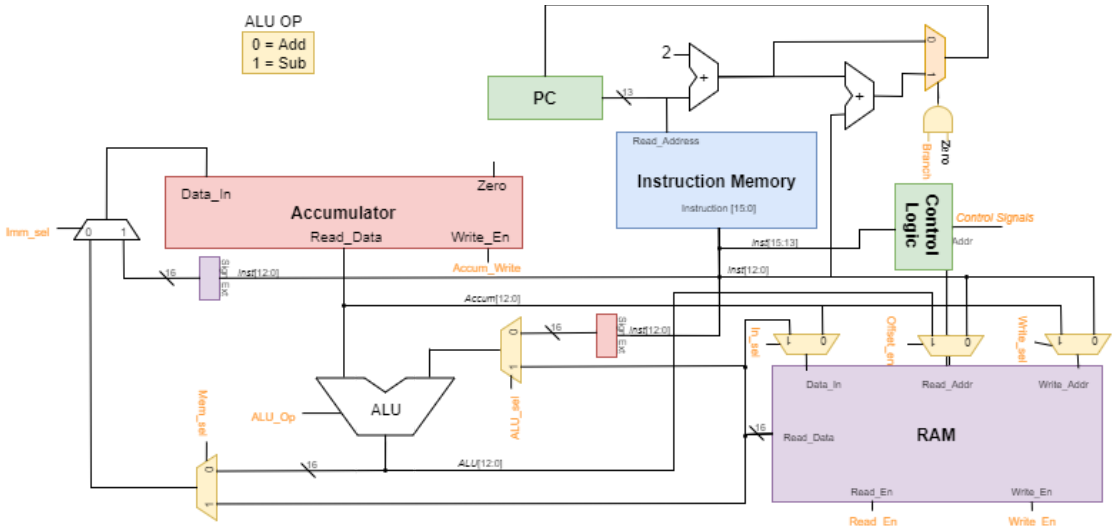

Figure One - MMARC-ONE Data Path

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.

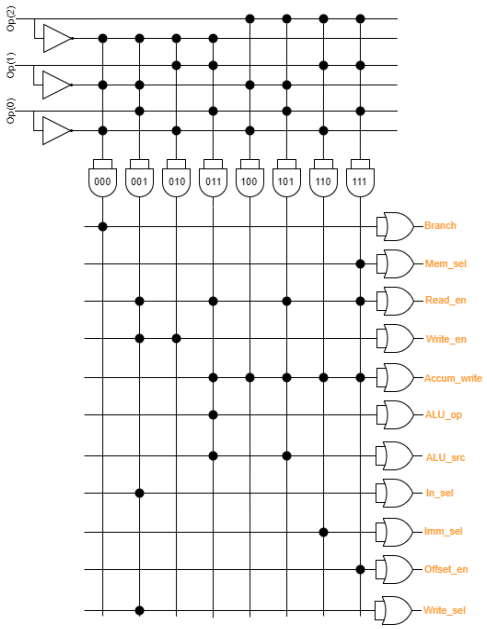

Figure Two - PLA

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.

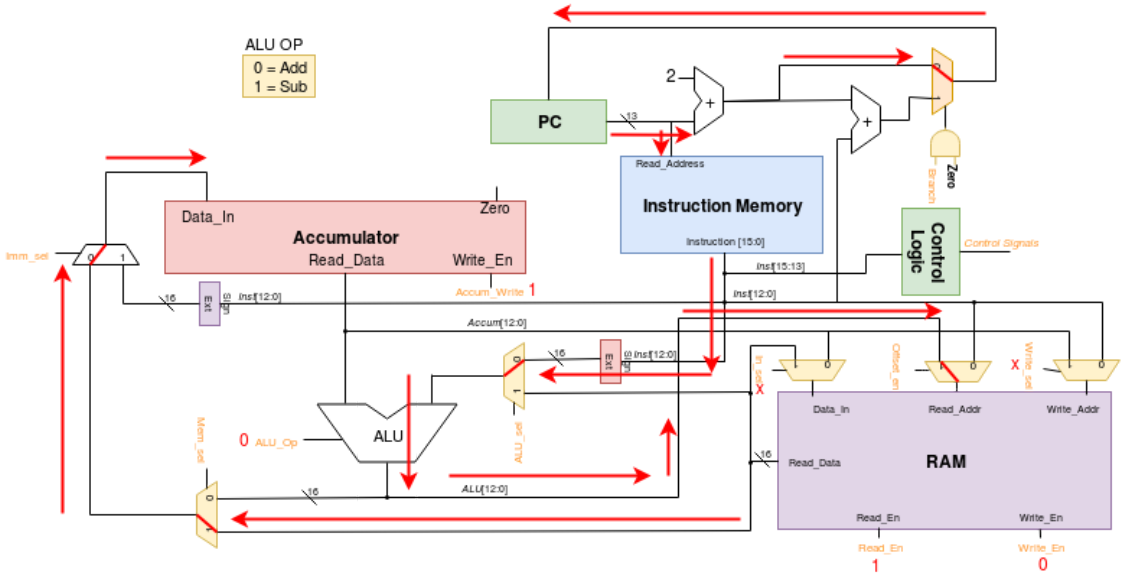

Figure Three - LAAD Instruction path

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
```