PureDarwin Documentation

Release 0.5

Benoit Allard

September 29, 2015

Contents

Contents:

CHAPTER 1

Introduction

puredarwin is a pure Hardware Implementation of CoreWar.

1.1 Darwin ???

Darwin is the predecessor of the CoreWar *game*.

1.2 CoreWar ?

(corewar.co.uk | corewar.info | corewars.org)

In Corewar, programs are fighting each other in the same memory space, the goal is to kill the other programs by making them execute illegal instructions.

Programs are written in assembly, RedCode is the name of the assembly used.

1.3 pure Hardware Implementation ?

Our world is made of two kind of people, the one that write software, and the one that design hardware. In the past, the former were slaved by the later, this is less and less true. Anyway, I made my study in the hardware field, and I'm now spending my day at $paying_job writing software.

Our goal here is to achieve a silicon running CoreWar game engine.

Actually, when I speak about an implementation of CoreWar, I am truly speaking about an implementation of MARS (**M**emory **A**rray **R**edcode **S**imulator) ... even if my implementation is everything but a simulator !

This implementation is split into Modules.

The code is written in an unexpected language when in comes to Hardware Design, I choose Python to help me with that task. Bare Python is of course not able to describe Hardware Modules, that's where MyHDL comes into the scene. MyHDL advertise itself as being able to output VHDL as well as Verilog, I'm curious of the result. Anyway, I don't own any FPGA, ASIC or even Lattice right now, so gtkwave will be my best guess when it comes to simulation for some time.

CHAPTER 2

Darwin

From Wikipedia:

> Darwin was a programming game invented in August 1961 by Victor A. Vyssotsky, Robert Morris Sr., and M. Douglas McIlroy. The game was developed at Bell Labs, and played on an **IBM 7090 mainframe** there. The game was only played for a few weeks before Morris developed an "ultimate" program that eventually brought the game to an end, as no-one managed to produce anything that could defeat it.

CHAPTER 3

RedCode

RedCode is the assembly language used to program the MARS Virtual Machine.

We will regard here only the the details we need to look at as hardware designer. The rest is left to the dozen of good tutorials you will find on the Internet. If you don't know where to start, Google might be your friend that time. That said, don't expect any easy talk in there.

They are different version of the standard defining the RedCode language, ranging from '86, '88 and extended ('94 has never been confirmed as a standard). We will try to implement most of the extended feature, while keeping compatibility with the previous standards.

3.1 Instruction Set

See RedCode Instruction Set.

3.2 Address Mode

First thing to kno. Is that memory access inside the MARS are relative to the current instruction pointer (noted IP here below). As seen from a program, all addresses are relative to the one of the currently executed instruction.

To keep a low memory footprint, RedCode has 5 address mode:

Table 3.1: RedCode address modes

Name	Relative operation	Absolute operation	A-Notation	B-Notation
Immediate Memory Addressing	0	IP	#	#
Direct Memory Addressing	x	IP + x	$	$
Indirect Memory Adressing	[IP + x]	IP + x + [IP + x]	*	@
Post Increment Indirect Memory Addressing	[IP + x]++	IP + x + [IP + x]++	{	<
Pre Decrement Indirect Memory Addressing	–[IP + x]	IP + x + –[IP + x/]	}	>

The last three one actually count double as we can address the first operand or the second operand on those addresses in Memory.

Out of those one, the last two one are pretty unusual, even for an experienced assembly programmer, just because of the fact that those Memory addressing modes *modify* the memory content instead of just pointing to it.

3.3 Examples

- IMP
- Dwarf
- Gemini

CHAPTER 4

RedCode Instruction Set

One feature of a processor running RedCode is that it has to implement some OS functionnalities at the silicon level. I'm mainly speacking here of the `SPL` instruction, and derivated ones. That instruction queue a new task in the task queue. For an unix developer, it is a *`fork(2)`* at the silicon level.

The typical example is:

```
SPL 0            ; execution starts here
MOV 0, 1
```

Since the `SPL` points to itself, after one cycle the processes will be like this:

```
SPL 0            ; second process is here
MOV 0, 1         ; first process is here
```

After both of the processes have executed, the core will now look like:

```
SPL 0            ; third process is here
MOV 0, 1         ; second process is here
MOV 0, 1         ; first process is here
```

So this code evidently launches a series of imps, one after another. It will keep on doing this until the imps have circled the whole core and overwrite the `SPL`.

CHAPTER 5

IMP

```
;redcode
;name Imp
;author A. K. Dewdney
;assert 1

     MOV      0,       1
```

CHAPTER 6

Dwarf

```
;redcode
;name Dwarf
;author A. K. Dewdney
;assert CORESIZE % 5 == 0

     DAT              -1
     ADD      #5,     -1       ; start address
     MOV      #0,     @-2
     JMP      -2
```

Another version put the Bomb after itself

```
;redcode
     ADD      #4,     3
     MOV      #2,     @2
     JMP      -2
```

Here is the result of a simulation on my MARS:

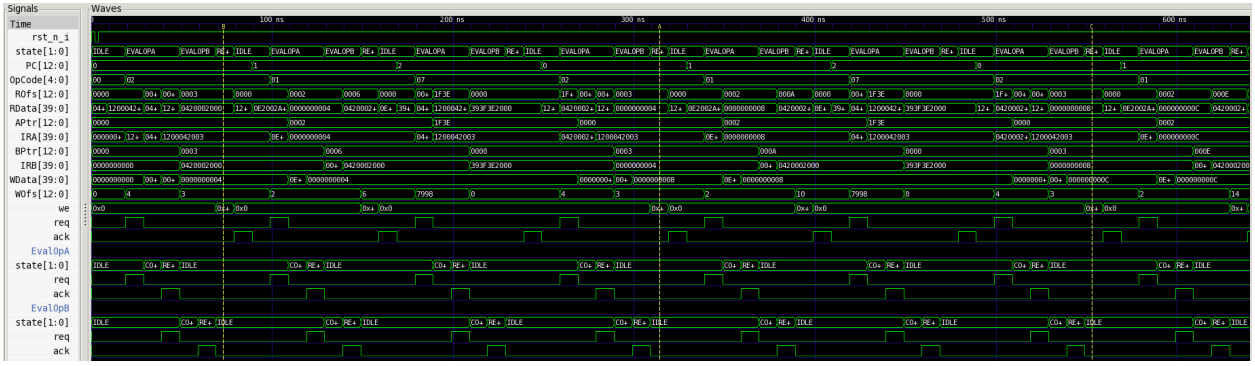


Fig. 6.1: Dwarf simulation

CHAPTER 7

Gemini

7.1 Code

```
;redcode
;name Gemini
;author A. K. Dewdney
;assert 1

      DAT               0
      DAT               99
      MOV       @-2,    @-1 ; start address
      SNE       -3,     #9
      JMP       4
      ADD       #1,     -5
      ADD       #1,     -5
      JMP       -5
      MOV       #99,    93
      JMP       93
```

7.2 Simulation

Fig. 7.1: Waveforms

CHAPTER 8

Modules

We have to split the design into modules

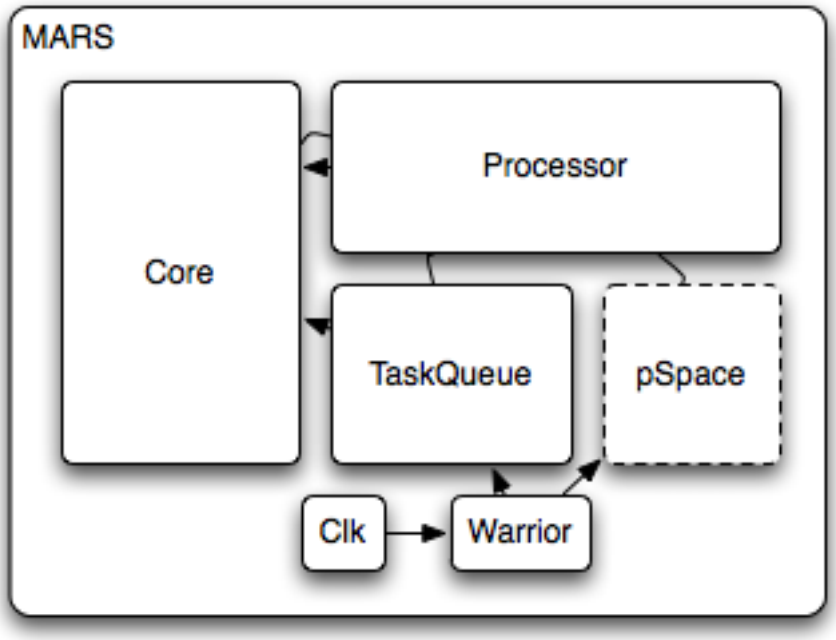


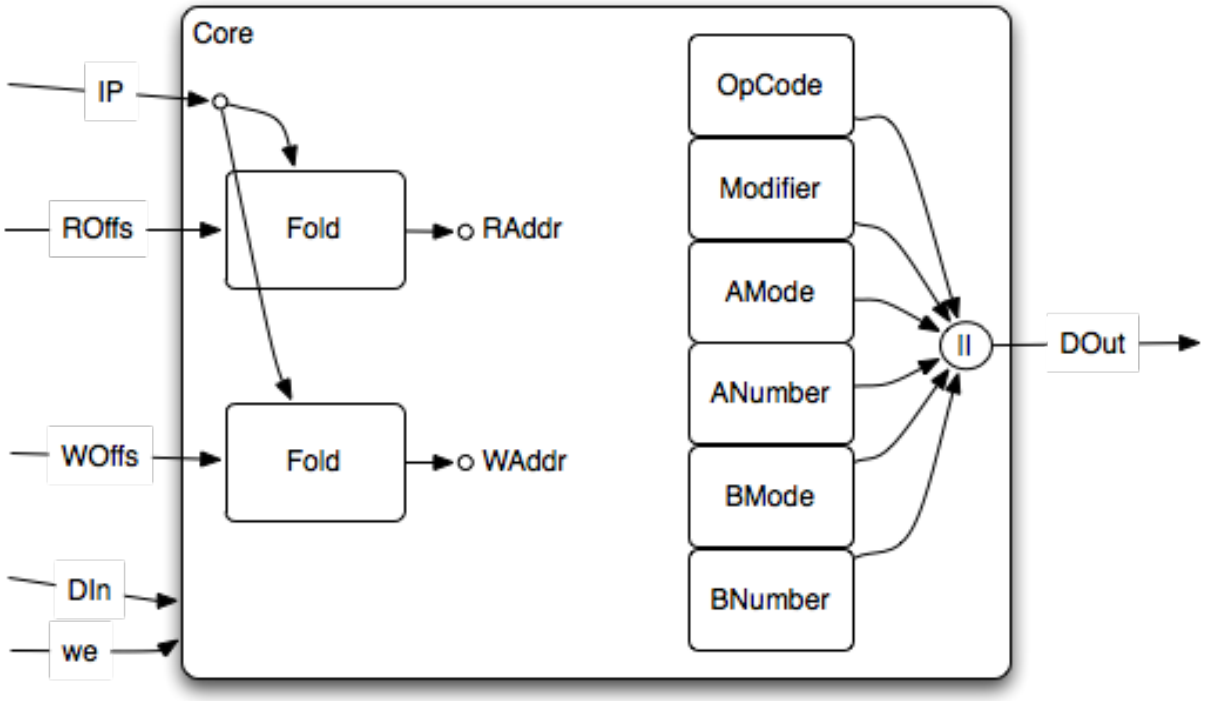
Fig. 8.1: Architecture

- The The Core alias the **Memory**
- The Task-Queue alias the **Scheduler** unit
- The The Proc alias the **Processing** unit
- The pSpace or the Process **private Storage** space

CHAPTER 9

The Core

The Core is the main **Memory** were the programs fight themselves. (As opposed to *each-others*, as nothing prevent you to hurt yourself.)



As we can see, the Core is split into RAM modules that each of them store a logical part of the Instructions. there are exactly **6** of them. One for each part:

1. OpCode
2. Modifier
3. AMode
4. ANumber
5. BMode
6. BNumber

9.1 Ports

9.1.1 input

Control Signals

- pc
- Wofs
- din
- ROfs
- we

Synchronous signals

- clk
- rst_n

Parameters

- maxSize: the depth of each internal RAM

9.1.2 output

As every good memory, we only have one output port being the data at the `ROfs` address.

- dout

9.2 Sub-processes

The Core has one main combinatorial subprocess that just split the incoming Instruction in chuncks for the RAM modules, and join back the chunks from the RAM modules into one Instruction for the others Modules.

9.3 internal Signals

Internal signals are only intercommunication signals between the sub-modules.

9.4 Submodules

9.4.1 RAM

See The RAM.

The basic component of the Core is a RAM with both a read and a write address bus. This will allow us to make asynchronous read, while making synchronous write.

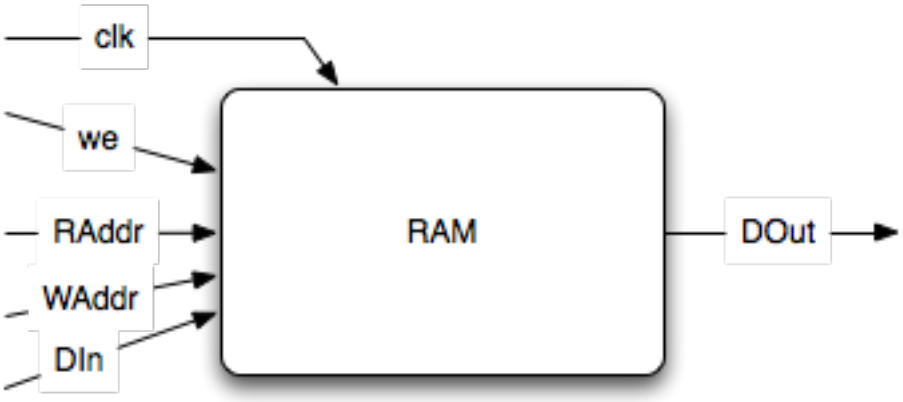
9.4.2 Fold

See Fold.

The Fold module just care about the fact that all our Read/Write are relative to the current Instruction Pointer. The Core itself, through this module takes offset as Input and translate those Offset as Absolute Addresses.

CHAPTER 10

The RAM



10.1 Ports

10.1.1 input

Control Signals

- raddr
- waddr
- din
- we

Synchronisation Signals

- clk
- rst_n (not taken into account in the current implementation)

Parameters

- width: the width (in bits) of the RAM
- depth: the depth (in number of cell) of the RAM

10.1.2 output

Control Signals

- dout

10.2 sub-process

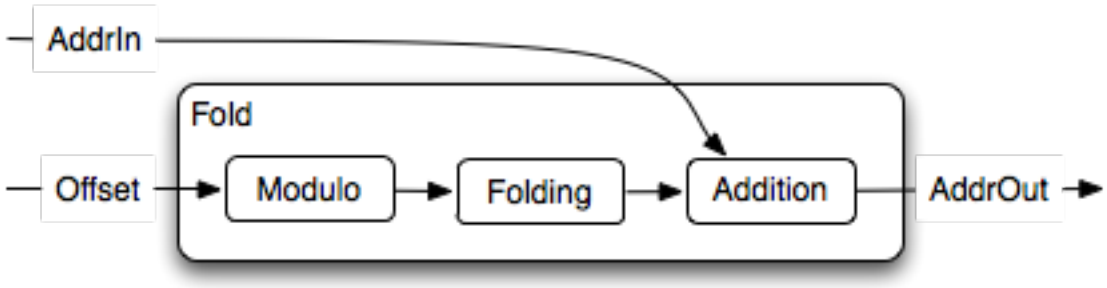
10.2.1 read

Read is completely combinatorial and simply return on `dout` the value of the RAM @ `raddr`.

10.2.2 write

Write is triggered by the clock, and write `din` to the RAM if `we` is set.

Fold



11.1 Ports

11.1.1 input

Control ports

- PC: The current Program Counter
- Offset

parameters

- limit: Folding limit
- maxSize: Size of the The Core

11.1.2 output

- Address

11.2 Sub-processes

This module is composed of three sub-modules:

- Modulo is used to calculate a temporary value
- Folding to Fold the Address into its Read (or Write) boundary.
- Addition to make the Address Absolute. (This step also includes a Modulo)

For the two first steps, the Cref provides the following listing:

```
/* There is one support function used to limit the range of */
/* reading from Core and writing to Core relative to the    */
/* current instruction.  Behaviour is as expected (a small  */
/* core within Core) only if the limits are factors of the  */
/* size of Core.                                            */

static Address Fold(
   Address  pointer,     /* The pointer to fold into the desired range.  */
   Address  limit,       /* The range limit.                             */
   Address  M            /* The size of Core.                            */
) {
   Address  result;

   result = pointer % limit;
   if ( result > (limit/2) ) {
      result += M - limit;
   };
   return(result);
}
```

11.3 Internal Signals

We have two internal signals to interconnect our three sub-process. Each one of them representing our output at its different stage of processing.

CHAPTER 12

Task-Queue

The task queue is a collection of `FiFos`. One for each Warrior keeping track of the current tasks running for each Warrior.

12.1 Sub-process

12.1.1 MUXs

One to direct the input to the right FIFO, another one to read the right task from the right FIFO.

12.2 Sub-modules

12.2.1 FIFO

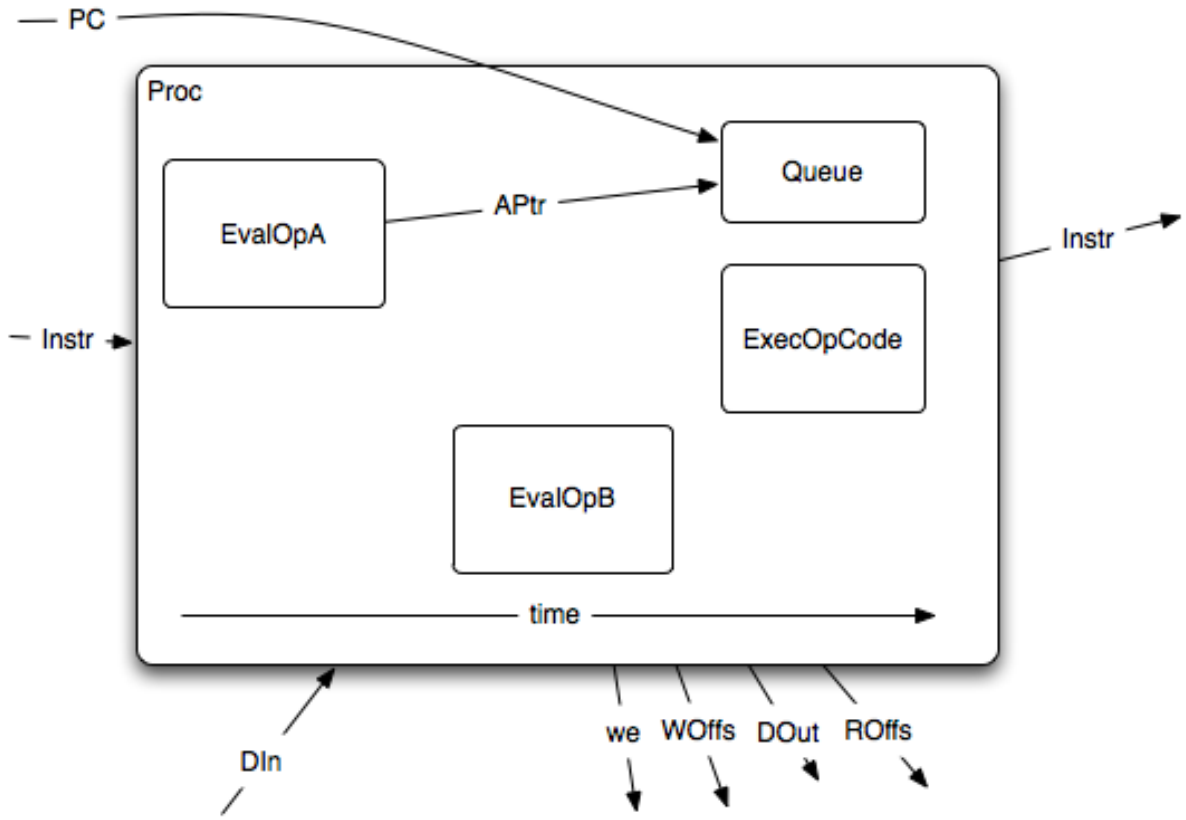
It is a FIFO that can queue two tasks in the same cycle.

One synchronous sub-process that reacts on `clk` and `rst_n`.

CHAPTER 13

The Proc

This is the Processing unit. Its main characteristic is that is has to be a state-less Module, as no state should subsist between processes. This same processing unit will be called by each of the Programs to try destroy each other. Every time, with a new instruction, and at the end of the processing, everything is done.



13.1 Ports

13.1.1 input

Control Signals

- Instr
- PC
- RData

Synchronous signals

- clk
- rst_n

Synchronisation signals

- req

13.1.2 output

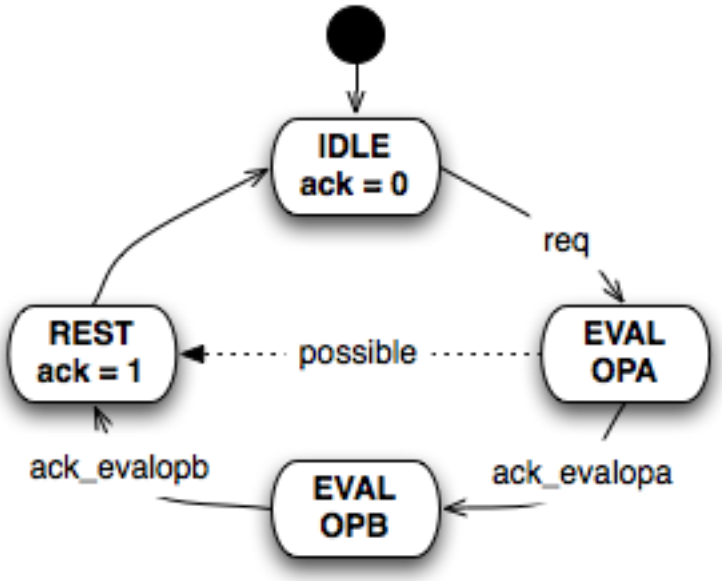
Control Signals

- IPOut1
- we1
- IPOut2
- we2
- WOfs
- WData
- we
- ROfs

Synchronisation signals

- ack

13.2 State-machine



13.3 Sub-processes

- link is actually just a splitter *à la* VHDL aliases.
- fsm is our FSM.

13.3.1 MUXs

the following process are just MUX that dispatch the info acording to the FSM state.

- fsmcore
- updateROfs
- updateIRX
- updatewe
- updatewdata

13.4 Sub-modules

13.4.1 EvalOp

Evaluate the operand part of the Instruction (Mode + Number). This one is instanciated twice, once for each of the Operands.

13.4.2 OutQueue

Make the Instruction readable for the Task-Queue.

13.4.3 OutCore

Make the Instruction readable for the The Core.

CHAPTER 14

pSpace

Not implemented

CHAPTER 15

ToDo

15.1 Loader

15.1.1 Description of the trouble

My main trouble at the moment is to find a way to Load the programms inside the Virtual Machine. The MARS had been designed as a **closed system**, with absolutely no input, and one output, namely the result of the *fight*.

That means that my current question doesn't get answered by design:

> How to get the programs in the Core of `puredarwin` MARS ?

I made a first try with an RS232 Module (self-designed) where the goal would be to load the Programs via a serial line ... I'm not entirely satisfied with that solution. Actually, I'm not sure that it would be practical.

15.1.2 Solution

Do not use any loader

For the moment, I would advice using `rev 548b16a67881` and traceProc.py as a main to simulate. This testModule also has a correct simulation implementation for the Queue and the Core. This shouldn't be a trouble to test a program on this MARS.

Quality of this solution

This solution makes some sense as the MyHDL module are actually not *directly* intended for synthesis. So, using simulation *tricks* in order to *simulate* is not seen as a betrayal.

CHAPTER 16

Status

`hg log` is your friend ...

16.1 Tue Mar 20 20:31:35 CET 2012

Reorganizing everything ...

16.2 Wed Dec 2 00:48:03 CET 2009

16.2.1 A good simulation basis

I realized the incapacity of MyHDL to synthesize anything. That's not bad in itself, Actually, I did not ordered my ALTERA yet, so, I'm not able to check any synthesis. And better yet, till now, even If I tried to keep to a synthetizable MyHDL, I never used it. I used MyHDL, (I have to admit it, without realizing it), as a prototyping tool, and thanks to it, I got results !

The current repository also has a good base for simulation, lots of components are there, pSpace is still missing, and some function implementation might also be missing, but a good basis is there.

Try it if you feel you can, and don't forget to report Issues !

CHAPTER 17

Indices and tables

- genindex
- modindex
- search