A09 Version 3

A09 is a very simple processor that represents a precursor to the X09. Version 3 is the most basic of the A09 series.

When the CPU is started it remains in a reset state until the Reset (Active low) pin goes high. The CPU then enters the S_Start1 state sequence where it loads the PC with a reset vector located at the bottom of memory = 0xFF.

ISA

Branch Control

- ~~Branch carry set (BCS) and clear (BCC)~~
- ~~Branch overflow set (BVS) and clear (BVC)~~
- Branch on equal (~~BEQ~~) and clear (**BNE**)
- ~~Branch on minus (BMI) and clear (BPL)~~
- ~~Branch on less than (BLT), (BGE), (BLE), (BGT)~~

Addressing modes:

- Immediate (value embedded in instruction) (LDI)
- Register to Register (MOV) Pseudo instruction. Implemented as: Dst ← Src + 0

Condition codes

Carry (C), Zero (Z), Negative (N), Overflow (V)

Instructions

1. NOP
2. HLT
3. ADD
4. SUB/CMP
5. SHR
6. SHL
7. BNE
8. LDI
9. JMP
10. RET
11. OTR
12. **MOV (pseudo instruction)**

13. JPL

NOP (0x01)

0	0	0	1												

- Immediate transition to Fetch cycle

ADD (0x02)

Dst ← S2 + S1, All registers must have been loaded prior.

0	0	1	0				Dst	Dst	Dst	S2	S2	S2	S1	S1	S1

IR[2:0] is routed to **Register Src 1** of the **Register-File**
IR[5:3] is routed to **Register Src 2** of the **Register-File**
IR[8:6] or IR[10:8] is routed to **Register Dest** of the **Register-File**

- **Instr** = 'b0 selects IR[8:6]
- **ALU_Op** = ADD
- **ALU_Ld** = low: Enable loading ALU results
- **FLG_Ld** = low: Enable loading ALU Flags

If IG = 0 then the 4th clock cycle is required. **CMP** instructions have no destination so IG = 1.

- **DATA_Src** = 'b10: selects **ALU** Results output register
- **REG_WE** = low: Enable loading the reg-file

SUB/CMP (0x03)

Dst ← S2 - S1, All registers must have been loaded prior.
CMP means compare as opposed to subtract. So "**SUB**" becomes "**CMP**" without needing a destination.
See **ADD** instruction above for the execution cycles and instruction format. They are identical.
IG = 1 means ignore "Dst", **IG** = 0 means store to "Dst"

0	0	1	1		IG		Dst	Dst	Dst	S2	S2	S2	S1	S1	S1

SHL (0x04)

Dst ← S1 << S2, All registers must have been loaded prior.
Shift S1 left S2 times with the results stored in Dst. Zero fill on LSB
See **ADD** instruction above for the execution cycles and instruction format. They are identical.

0	1	0	0				Dst	Dst	Dst	S2	S2	S2	S1	S1	S1

- **ALU_Op** = SHL

SHR (0x05)

Dst ← S1 >> S2, All registers must have been loaded prior.
Shift S1 right S2 times with the results stored in Dst. Zero fill on MSB
See **ADD** instruction above for the execution cycles and instruction format. They are identical.

0	1	0	1				Dst	Dst	Dst	S2	S2	S2	S1	S1	S1

- **ALU_Op** = SHR

BNE (0x06)

PC ← [A7:A0] - WordSize
Branch is taken if the Z=0
A(7:0) = specifies signed branching address relative to PC.

0	1	1	0					A7	A6	A5	A4	A3	A2	A1	A0

IR[7:0] is routed to **Sign-Extend** and then to “Add-Wordsize” then routed to **MUX_PC**
If branch is taken:

- **BRA_Src** = ’b**01**: select Sign-extend source
- **PC_Src** = ’b**00** which selects Branch-Address source. PC has advanced prior so it must be decremented
- **PC_Ld** = low: enables loading PC
- **FLG_Rst** = low. Flags are needed anymore so reset them.

JMP (0x07)

PC ← [S1]
S1’s content specifies an absolute address to jump to. S1 must be loaded prior.

0	1	1	1										S1	S1	S1

Register Dest = IR[2:0] is routed to **Register-File**’s decoder for enabling LD for the destination register

- **PC_Src** = ’b**001**: Select Reg-file source 1
- **PC_Ld** = high: Disables loading of PC

RET (0x08)

PC ← [Stack]
The Stack’s content is transferred to the PC.

1	0	0	0												

- **PC_Src** = ’b**01**: Selects Stack Top for return address

- **PC_Ld** = low: Enables loading of PC

LDI (0x09)

Dst ← #(V7:V0)
V(n) is loaded into the destination register Dst(n).

1	0	0	1		Dst	Dst	Dst	V7	V6	V5	V4	V3	V2	V1	V0

IR[10:8] is routed to **MUX_DATA** (**Register Dest**) which specifies the destination register.

- **Instr** = ‘b1 selects IR[10:8]
- **REG_WE** = low: If High then none of the registers are enabled for loading
- **DATA_Src** = ’b00: selects zero-extended as the source

OTR (0x0A)

Copy data from memory to a ½ Word size output register. The register’s output is typically connected to some sort of output pins.

1	0	1	0										S1	S1	S1

- **Out_Ld** = low: enable loading output
- **Out_Sel** = low: Select reg-file source

HLT (0x0B)

1	0	1	1												

- **Halt** = ‘b1 forces CPU to stop
- **Ready** = 1’b0: signals CPU is no longer ready
- next_state = S_HALT

JPL (0x0C)

PC ← [S1]
S1’s content specifies an absolute address to jump to. S1 must be loaded prior.
PC+1 is stored on the Stack

1	1	0	0										S1	S1	S1

Fetch cycle

The fetch cycle retrieves an **Instruction** from memory and loads it into the **IR** register.

S_Fetch

- **IR_Ld** = low: enable loading of IR
- **PC_Inc** = low: enable incrementing PC

Sequence Controller

The sequence controller (matrix) job is to manipulate the control signals that fetch and handle data flow. There are 3 or 4 Clocks to an instruction cycle: two for *fetching* and 1 or 2 for *execution*.

Mux / Demux

https://verilogcodes.blogspot.com/2015/10/verilog-code-for-14-demux-using-case.html

Memory

- https://stackoverflow.com/questions/41499494/how-can-i-use-ice40-4k-block-ram-in-512x8-read-mode-with-icestorm
- https://www.reddit.com/r/FPGA/comments/8y91kv/bram_issues_with_ice40/ Talks about the “**MODE**”s, for example, MODE0 = 256x16.
- Memory is simply BRAM: https://zipcpu.com/tutorial/lsn-08-memory.pdf
- https://github.com/Megamemnon/bram
- https://verilogguide.readthedocs.io/en/latest/verilog/designs.html#random-access-memory-ram

If you want a block RAM, you need to follow certain rules:

1. Any RAM access should be contained in its own always block
2. RAM can only be initialized **once**
3. Don’t put RAM access in a cascaded if
4. Don’t put RAM in a port list
5. Don’t put RAM in a block with other things. See #1

```
#1 above
always @(posedge i_clk)
     if (write)
          ram [write_addr] <= write_value;

always @(posedge i_clk)
     if (read)
          read_value <= ram[read_addr];
```

RAM

```
reg [W-1:0] ram[0:(1<<SIZE)-1];
```

Lattice ice40

BRAM bit-stream config issue: apparently you need ~3us prior to accessing:
https://twitter.com/wren6991/status/1363518292036583429?s=20

https://github.com/m-labs/nmigen/blob/4a59d5d1797c3c39b8f08d0ed9e2abbf71739cda/nmigen/vendor/lattice_ice40.py#L338

```
# For unknown reasons (no errata was ever published, and no documentation mentions this
        # issue), iCE40 BRAMs read as zeroes for ~3 us after configuration and release of internal
        # global reset. Note that this is a *time-based* delay, generated purely by the internal
        # oscillator, which may not be observed nor influenced directly. For details, see links:
        #  * https://github.com/cliffordwolf/icestorm/issues/76#issuecomment-289270411
        #  * https://github.com/cliffordwolf/icotools/issues/2#issuecomment-299734673
        #
        # To handle this, it is necessary to have a global reset in any iCE40 design that may
        # potentially instantiate BRAMs, and assert this reset for >3 us after configuration.
        # (We add a margin of 5x to allow for PVT variation.) If the board includes a dedicated
        # reset line, this line is ORed with the power on reset.
        #
        # The power-on reset timer counts up because the vendor tools do not support initialization
        # of flip-flops.
```

Issue: Yosys keeps thinking one or the other of my arrays should be implemented using logic cells.

Solution: I was crossing clock domains when I connected the two modules so needed to synchronize reading and writing from the array. The lack of coordination between reads and writes to the array was causing yosys to infer that the array was not to be implemented as block ram.

Single-Port RAM

```
module ram (din, addr, write_en, clk, dout);// 512x8
   parameter addr_width = 9;
   parameter data_width = 8;

   input [addr_width-1:0] addr;
   input [data_width-1:0] din;
```

```
    input write_en, clk;
    output [data_width-1:0] dout;

    reg [data_width-1:0] dout; // Register for output.
    reg [data_width-1:0] mem [(1<<addr_width)-1:0];

    always @(posedge clk)
    begin
       if (write_en)
          mem[(addr)] <= din;
       dout = mem[addr]; // Output register controlled by clock.
    end
endmodule
```

ROM

- On-chip RAM can only be initialized in an initial block.
- Cannot re-initialize a block RAM in this fashion later without reconfiguring (i.e. reloading) the FPGA.

```
#2 above

reg [31:0] ram[0:256];
initial $readmemh (FILE_NAME, ram);
```

Notes

Draw.io data flow diagram:
https://app.diagrams.net/#G1IysipHvNSD_dchEhOhqFgbAMjjcquNPM
An excellent RISC-V 32I yet very minimal:
https://github.com/BrunoLevy/learn-fpga/blob/master/FemtoRV/RTL/PROCESSOR/femtorv32_single_file.v

Verilog will gladly truncate upper bits, without warning. Originally I accidentally merged the reset vector states with the CPU states and because of that the upper 2 bits were thrown away thus preventing the reset sequence from reaching S_Vector4 = ‘b0100. Instead it would wrap back around to an invalid state of ‘b00. Because I defined the reset vector states as 2 bits this resulted in the upper two bits truncated away.

For example:
```
localparam  S_Reset          = 4'b0000,
           S_Vector1        = 4'b0001,
           S_Vector2        = 4'b0010,
           S_Vector3        = 4'b0011,
           S_Vector4        = 4'b0100,    ← We can’t reach this state
           S_FetchPCtoMEM   = 4'b0101,
           S_FetchMEMtoIR   = 4'b0110,
           S_Decode         = 4'b0111,
           S_Execute        = 4'b1000,
           S_Ready          = 4'b1001,
           S_ALU_Execute    = 4'b1010, // Extra cycle for ALU instructions
           S_HALT           = 4'b1011;
```

The fix was to do what I should have done in the first place and that was to separate the two state spaces.

```
localparam  S_Reset         = 4'b0000,
           S_FetchPCtoMEM   = 4'b0001,
           S_FetchMEMtoIR   = 4'b0010,
           S_Decode         = 4'b0011,
           S_Execute        = 4'b0100,
           S_Ready          = 4'b0101,
           S_ALU_Execute    = 4'b0110, // Extra cycle for ALU instructions
           S_HALT           = 4'b0111;

localparam  S_Vector1       = 2'b00,
           S_Vector2        = 2'b01,
           S_Vector3        = 2'b10,
           S_Vector4        = 2'b11;
```

Another truncation example is pc_src, it is defined at one location as 3 bits yet sequence_control defines it as 2 bits. This is another truncation issue that the simulation gladly truncates. This resulted in the improper PC source being selected. This was the issue:

```
reg [1:0] pc_src;       // MUX_PC selector
```

The correction is to carry through the bit size properly:

```
reg [PCSelectSize-1:0] pc_src;       // MUX_PC selector
```

Build command

```
set -e [ -n "$NMIGEN_ENV_IceStorm" ] && . "$NMIGEN_ENV_IceStorm"
: ${YOSYS:=yosys} : ${NEXTPNR_ICE40:=nextpnr-ice40} :
${ICEPACK:=icepack} "$YOSYS" -q -l top.rpt top.ys
```

```
"$NEXTPNR_ICE40" --quiet --log top.tim --hx8k --package tq144:4k
--json top.json --pcf top.pcf --asc top.asc "$ICEPACK" top.asc
top.bin
```

References

Search term: “8-bit multicycle processor design verilog code”

- https://github.com/axel223/8-bit-cpu-in-verilog
- https://courses.cs.washington.edu/courses/cse378/10sp/lectures/lec08-singlecyc.pdf single cycle 8bit cpu
- https://courses.cs.washington.edu/courses/cse378/10sp/lectures/ all the lectures of 8bit single cycle cpu
- Nihongo/Storage/Hardware/FPGA/TN1250 - Memory Usage Guide for iCE40 Devices pdf
- https://github.com/ellore/processor Implementation of 8-bit multi cycle processor using verilog on FPGA
- https://zipcpu.com/tutorial/lsn-08-memory.pdf
- https://www.fpga4student.com/2017/01/verilog-code-for-single-cycle-MIPS-processor.html
-

Books (Nihongo/storage/Hardware/Verilog)
- Computer Architecture: A Quantitative Approach
- Verilog HDL Synthesis A Practical Primer
- Digital Systems Design Using Verilog by Charles Roth, Lizy K. John, Byeong Kil Lee 2016 MIPS CPU
- Computer Architecture Tutorial Using an FPGA: ARM & Verilog Introduction ← Covers **generators**. A multiplexer using a generator.