8-Bit Softcore CPU

Submitted as the Capstone Project for the Master of Engineering Degree

Max Thrun

April 9, 2014

1 Abstract

The purpose of this project is to create a custom 8-bit softcore computer processing unit (CPU) which can ultimately be used and expanded on for both educational and practical purposes. This project explores the design and implementation of a complete system on chip (SoC) in the hardware description language Verilog and realized on a field programmable gate array (FPGA). A simple assembler and program loader, written in Python, are also developed in order to form a complete development package. With these components we are able to compile programs written in our custom assembly language and run them on our custom CPU in a FPGA development board.

2 Introduction

The understanding, design, and implementation of a computer processing unit is something that is often challenging for most people when they are first studying the field. There are seemingly infinite resources on the subject but there are very few that provide an easily digestible top to bottom example going from the compiler down to the hardware.

When I personally started studying the topic of CPUs I had a strong urge to implement my own and be able to compile and run small programs on it, which is a sentiment I feel is common amongst people first getting interested in the topic. Like many, I have found that I often learn best by example and I was frustrated with the lack of easy to understand example projects out there. It was not until I came across the Your First CPU! [1] blog posts, which provided a simple concrete example of implementing a CPU, that things really clicked. The Your First CPU! blog posts were good at helping me get an understanding of a CPU on the hardware level but it left a lot to be desired on the software end as it provided fairly complex Flex and Bison [2] scripts to parse and assemble programs. While these tools are popular and extremely powerful I felt that they were over complicated for what I wanted to achieve at the time which was to just simply compile a small program and run it on my custom CPU.

There are numerous other example projects out there of CPUs but most that I have come across in the past were either too complex, incomplete, or just implemented in a way that I personally found confusing. With this project I try to bridge some of these gaps by providing a small, simple, working example of a CPU as well as providing a basic assembler and a tool to load and run programs.

3 Implementation

3.1 Design Overview

The main goal of this project is to show an example of a working CPU. While this can be achieved purely through simulation it is much more satisfying to see a physical implementation running on a development board which can provide various inputs and outputs (IO). For my implementation I chose to use an Altera DE-1 development board which has a Cyclone II FPGA and a plethora of various IO such as LEDs, switches, 7 segment displays, as well as more complex peripherals such as audio.

In order to communicate with these peripherals we need additional components besides the CPU. Firstly, we need random access memory (RAM) which we will use to store our program and which the program will use to store information while it is running. The Cyclone II, like most modern FPGAs, has internal RAM which we can use for this purpose. Secondly, we need some general purpose input and output (referred to as GPI and GPO in this project) modules in order to interface with peripherals such as LEDs and switches. Additionally, we would like to have some way of performing timed events, such as flashing the LEDs a certain rate. A 32-bit timer module has been implemented to achieve this. When the CPU needs to read or write data we need module which can determine if the CPU is trying to talk to RAM or a peripheral. This module is called a memory mapper as it maps different memory locations as seen by the CPU to various other components throughout the system. Finally, we need a way to load the programs into the system which is achieved by a serial bootloader.

A system block diagram which illustrates the connection between the DE-1 peripherals and the various components in the design is shown below.

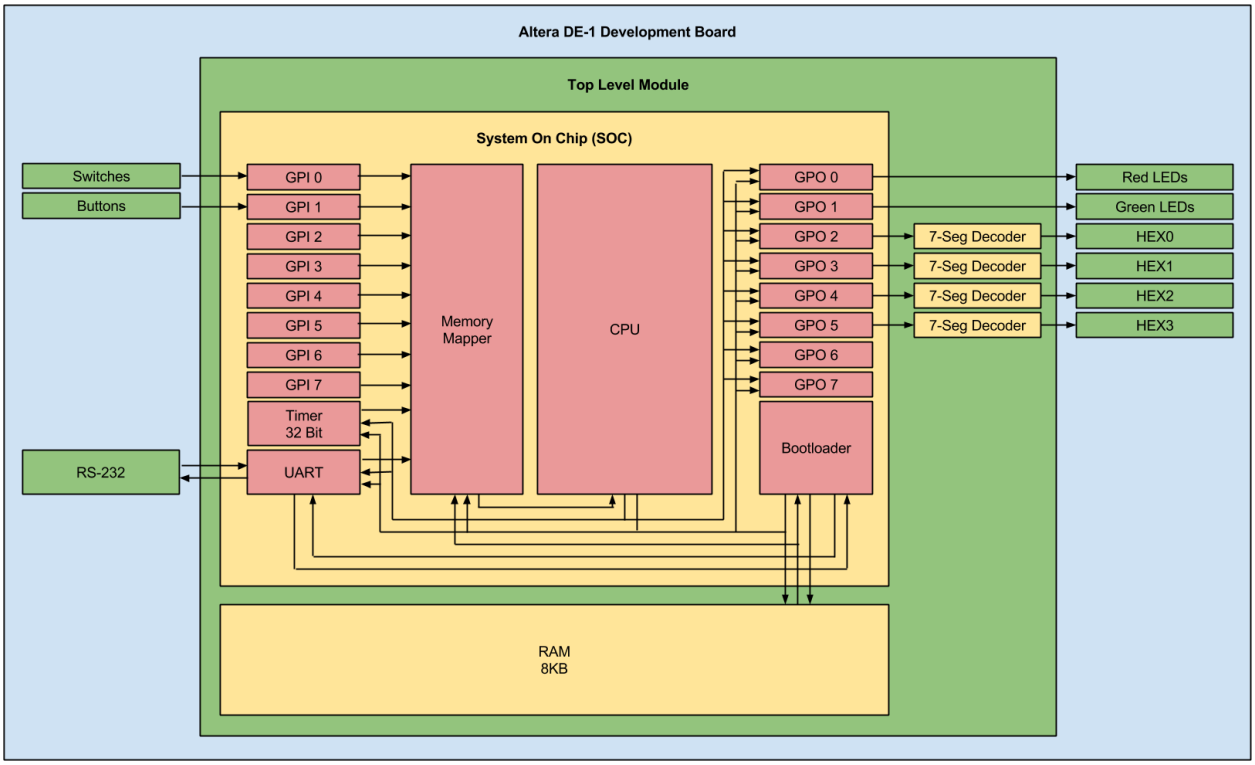

Figure 1: System Block Diagram

3.2 CPU

The CPU for this project is non-piplined and each instruction takes 7 clock cycles to complete. Additionally, I have chosen to go with a Von Neumann style architecture [4] where the data and instructions are both stored in the same memory and accessed by a single bus. The reason for this over a Harvard style architecture [6] is that it is slightly simpler to implement and also allows us to more easily achieve cool tricks such as having a program modify its own code [5]. One of the down sides of using a Von Neumann architecture is that you cannot access the instruction and data memory at the same time. This means that you would typically need a whole extra cycle just to fetch or store data. To avoid this I am actually clocking the CPU on the opposite edge of the clock relative to the rest of the system. This means that the CPU steps on the positive edge and the memory responds on the negative edge which allows the data to be ready by the next CPU cycle. Doing this saves us up to 2 clock cycles per instruction.

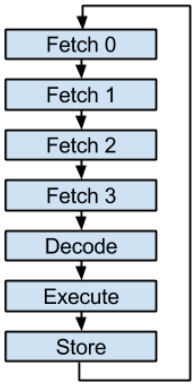

Figure 2: CPU Instruction Cycle

Each instruction is 4 bytes long which requires 4 fetch stages to form the complete instruction word as our RAM only accesses single bytes at a time. Typically, the next stage would be to decode the instruction being that in some instruction sets the operands might cross byte boundaries and you need to wait until you have fetched all the bytes before you can decode the instruction. To simplify things, in our CPU the opcode and each operand get their own separate byte with the exception of memory addresses which require two bytes. The side effect of this is that each operand can address up to 256 registers which in practice is unusual (a typical CPU might only have a couple dozen registers at most). We will consider this huge register file a feature as it actually makes writing a high level language compiler significantly easier as you do not need to worry about high pressure register allocation. Since we do not actually have to decode the instruction the decode stage is used to increment the program counter and switch the memory address that the CPU is looking at to the address of the data specified by the instruction.

The table below summarizes the 22 instructions that are currently implemented.

Mnemonic	Operands	Description	32-Bit Instruction Word MSb LSb
HALT		Stop program execution	`00000000 -------- -------- --------`
LD	`d,m`	Load value at address `m` into register `d`	`00000001 dddddddd mmmmmmmm mmmmmmmm`
ST	`s,m`	Store register `s` into memory address `m`	`00000010 ssssssss mmmmmmmm mmmmmmmm`
STL	`l,m`	Store literal `l` into memory address `m`	`00000011 llllllll mmmmmmmm mmmmmmmm`
LDL	`d,l`	Load literal `l` into register `d`	`00000100 dddddddd aaaaaaaa --------`
MOV	`d,s`	Move register `s` into register `d`	`00000101 dddddddd aaaaaaaa --------`
ADD	`d,a,b`	`d = a + b`	`00000110 dddddddd aaaaaaaa bbbbbbbb`
SUB	`d,a,b`	`d = a - b`	`00000111 dddddddd aaaaaaaa bbbbbbbb`
AND	`d,a,b`	`d = a & b`	`00001000 dddddddd aaaaaaaa bbbbbbbb`
OR	`d,a,b`	`d = a | b`	`00001001 dddddddd aaaaaaaa bbbbbbbb`
XOR	`d,a,b`	`d = a` $\oplus$ `b`	`00001010 dddddddd aaaaaaaa bbbbbbbb`
SFL	`d,a,b`	`d = a << b`	`00001011 dddddddd aaaaaaaa bbbbbbbb`
SFR	`d,a,b`	`d = a << b`	`00001100 dddddddd aaaaaaaa bbbbbbbb`
INC	`d,a,b`	`d = a + 1`	`00001101 dddddddd aaaaaaaa bbbbbbbb`
DEC	`d,a,b`	`d = a - 1`	`00001110 dddddddd aaaaaaaa bbbbbbbb`
EQL	`d,a,b`	`d = a == b`	`00001111 dddddddd aaaaaaaa bbbbbbbb`
GTH	`d,a,b`	`d = a > b`	`00010000 dddddddd aaaaaaaa bbbbbbbb`
LTH	`d,a,b`	`d = a < b`	`00010001 dddddddd aaaaaaaa bbbbbbbb`
INV	`d,a`	`d = ~a`	`00010010 dddddddd aaaaaaaa --------`
BRZ	`a,m`	Branch to `m` if register `a` is zero	`00010011 aaaaaaaa mmmmmmmm mmmmmmmm`
BRNZ	`a,m`	Branch to `m` if register `a` is not zero	`00010100 aaaaaaaa mmmmmmmm mmmmmmmm`
JMP	`m`	Jump to address `m`	`00010101 mmmmmmmm mmmmmmmm --------`

Table 1: Instruction Set Description & Encoding

3.3 Memory Mapping

All input, outputs, and internal peripherals such as the 32-bit timer and the UART are accessed and controlled via memory accesses in a scheme called memory-mapped IO [7]. In this scheme each peripheral is assigned a memory address and then watches for this address on the memory bus. When the CPU writes to this address it stores the data to its own internal registers. When the CPU is doing a read we need a single entity in charge of figuring out which peripherals data to give to the CPU. This entity is called the memory mapper and it has a list of the memory addresses of all the peripherals. All the peripherals feed their outputs into the memory mapper. The memory mapper then looks at the address that the CPU wants and feeds it the corresponding peripherals data. If the requested address does not belong to any peripheral the memory mapper selects the RAM as the CPU input.

One downside of this approach is that we are wasting RAM. When the CPU writes to the memory address belonging to a peripheral that data also gets written into RAM but when the CPU reads back the memory at that address the memory mapper will direct the peripherals output to the CPU, not the RAMs output. In our case we have only a handful of memory mapped peripherals and lots of RAM so it is not a huge issue.

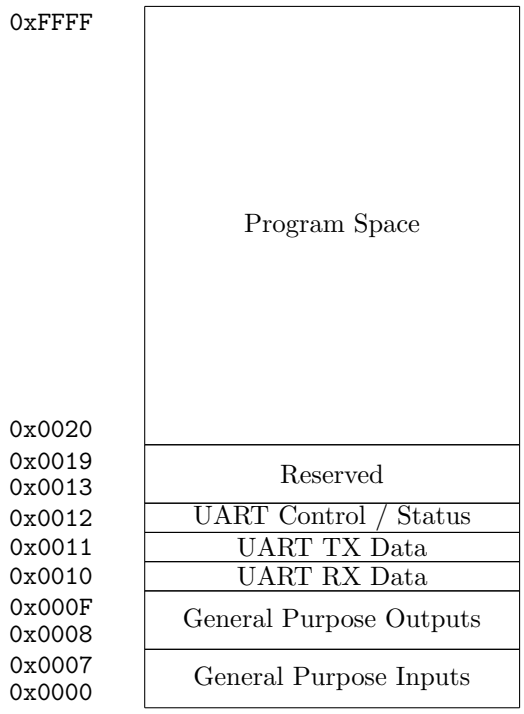

Figure 3: SoC Memory Map

The diagram to the right shows the memory map of our address space. There are 8 GPI peripherals and 8 GPO peripheral and each one takes 1 byte. The UART uses 3 bytes, 2 for data in and out and a third one for status and control. Addresses `0x0013` through `0x0019` are reserved for future peripherals that might be added. The program space starts at `0x0020` and extends all the way to `0xFFFF` which gives the program about 65KB of usable space (not accounting for the size of the program itself). Note that this is the maximum amount of memory that the CPU is capable of addressing, it does not mean that amount of RAM will actually be available. The amount of RAM in FPGAs varies so the size of the RAM is configurable at build time by setting the `RAM_ADDR_BITS` define in the `constants.v` file. In my case, I can only fit about 8KB of RAM in the Cyclone II that is on my development board. Most actual production SoCs contain internal RAM but since I am targeting FPGAs, which might not have a lot of internal RAM, I did not want to make internal RAM a hard requirement so I brought out the RAM interface. This makes it easy to use re-use the SoC module with external RAMs, such as the SRAM that is on my DE-1 development board.

3.4 Bootloader

One of the bigger challenges of running our system on an FPGA is loading the program into RAM. Under simulation it is as easy as using the `$readmemb("filename")` function but it is not as straight forward when targeting a physical device. It is possible to bake the program in with the FPGA bitstream using a memory initialization file [8] but this would require us to resynthesize and flash the FPGA every time we wanted to run a new program.

Another obvious solution would be to store the program on an SD card and write a hardware state machine to read it. While this initially seems like an attractive solution due to the ubiquity of SD cards it has three major drawbacks: interfacing with an SD card is not simple, not all FPGA development boards have an SD card slot, and every time you want to update the program you have to remove the SD card, put it in your computer, change the program, take it out, put it back in the dev board, and reset the FPGA. For rapid development, which is what we want to do, this process is unacceptable.

Yet another solution is to load our program via a serial port. All FPGAs are capable of implementing a UART and getting a USB to UART adaptor is cheap and easy. Also, with this method we can simply press a button on the development board to put the system into 'boot mode' and then run a little script on the computer to send the program over. The only disadvantage of this method is that the program is volatile, if we power cycle the board we have to resend the program. Since the goals of this project do not involve embedding our FPGA into a system that needs to be able to survive power cycles the UART solution is great.

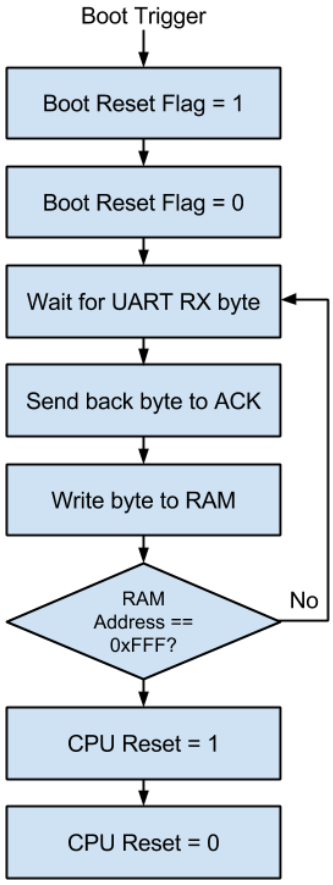

Figure 4: Bootloader State Machine

The UART bootloader needs to be able to do 2 main things: receive and write a program to RAM, and reset the CPU after it is done. The state diagram shown above illustrates this process. The state machine sits in an idle state until a boot trigger event, which I have mapped to a button on my development board. It then drops through two states which pulse the boot reset flag. This flag is used to reset the UART to clear any status flags that may be set. The bootloader also asserts a signal to indicate that it is in boot mode. This switches the RAMs address and data lines from the CPU to the bootloader. The bootloader then continuously reads bytes from the UART and loads them into RAM, incrementing the RAM address after each byte, until it reaches the top of the RAM. The disadvantage of this method is that we have to send the full RAM contents, even if it is mostly zeros. This is significantly slower than just sending the RAM contents that the program actually occupies. A possible solution to this might be to run length encode the data but with small RAM sizes (like the one in our system) it only takes a few seconds to load so the effort of speeding it up is probably not worth it. Once the bootloader has determined that it has received all the bytes it puts the CPU in reset and de-asserts the boot mode flag. It then de-asserts the CPU reset line and execution begins. It is important to have the CPU in reset before de-asserting the boot mode flag as once the flag is de-asserted the CPU has control over the RAM. Since the contents of RAM were unknown and changing during boot the CPU could be executing anywhere. We don't want to give control of the RAM back to the CPU until we reset it.

3.5 Assembler

To construct a program that can run on our CPU we simply need to lookup the binary encoding of the instruction that we want to use. While it is certainly possible to assemble a program by hand it is much less tedious to have an assembler do it for you. I have created a simple assembler (`./tools/asm.py`) which supports some of the typical features one would expect from an assembler.

The assembler reads the input `.asm` file line by line and first strips the line of comments and forces it to upper case. It then figures out if the line is a directive (like `.define`), an instruction, or a label. If it is an instruction it forms the 4 byte instruction word (using the OP code that it automatically parses out of the Verilog `constants.v` file) and adds those bytes to an array. If it comes across a label it records the name of the label and what byte location it is at. When it comes across a jump it records what the destination label is and then fills in zeros for the address since it might not know what the address of the label is yet (if it wants to jump forward in the program we do not yet know the address of that label). After all instruction bytes have been added to the array it goes back through all the jumps, looks up the position of the label it wants to jump to, and fills out the address of the jump instruction. It then writes all the bytes out to two files: one file which is a raw binary file that we will load into the FPGA (called a ROM), and another which is an ASCII file formatted in a way that can be read in by the Verilog simulator.

Other features such as aliases, the equivalent to `#define` in C, are supported through the `.define` directive and comments are fully supported as well. The assembler also parses the `constants.v` file to figure out the memory location of all the memory mapped peripherals and automatically adds an alias for them. This is nice since when you add a new peripheral you only have to update the Verilog file and the assembler automatically will account for it. The whole assembler is only about 150 lines of code, most of which is just putting the operands in the correct byte order for each instruction.

An example program which increments the value shown on the LEDs every time the timer goes off is shown below.

```
1  .define LEDR GPO_0   ; red leds are on GPO 0
2  .define LEDG GPO_1   ; green leds are on GPO 1
3
4  ldl r0,0             ; initialize register 0 to 0
5
6  main:
7      st r0,LEDG       ; output register 0 to green leds
8      st r0,LEDR       ; output register 0 to red leds
9      inc r0           ; increment register 0
10
11     stl 0,TMR_RST    ; reset the timer by writing to the TMR_RST address
12
13 delay:
14     ld r1,TMR_TRIG   ; get the status of the timer trigger
15     brz r1,delay     ; if its still zero keep delaying
16
17     jmp main         ; go back and do it all again
```

Listing 1: Count up on the LEDs

4 Results

The results of this project are best conveyed through video. The following videos show the CPU running four different test programs on the DE-1 development board.

Test Program	Description	Video Link
`shift.asm`	Shifts a single LED down both sets of LEDS	`http://youtu.be/aNCSzHn3NVQ`
`count.asm`	Counts up in binary on the LEDs	`http://youtu.be/9p6P2v8b_BA`
`seg_count.asm`	Counts up in base 10 on the 7-segment displays	`http://youtu.be/zjMZnJXbdMo`
`blink_1hz.asm`	Blinks the green LEDs at 1hz	`http://youtu.be/qbEIPBlEjM4`

We can see from the videos above that the CPU and supporting components all seem to be working as expected. While these small test programs are no where near that complexity of what a typical application might be like it at least proves that the system is working at a fundamental level and that all of our development tools are working correctly. With just 2 lines of Verilog and 4 lines of Python we can add additional instructions to the CPU and start exploring more complex topics and designs in the future. Given the ease in which we can modify and extend this project I believe we have achieved the original goal of building a complete example system from the CPU all the way to the compiler.

5 Discussion

I believe the efficacy of my implementation is strong given the overall success of the project. The original goal was to produce a small and simple example CPU and compiler that can be used primarily as a learning platform. While its effectiveness in conveying knowledge is yet to be determined the project is completely functional from a technical aspect. I believe I was able to keep it relatively simple while still achieving interesting features such as the 32-bit timer and serial bootloader.

There are a few things I felt I have learned from this project. From a technical perspective I formed an appreciation for how much work goes into the details of CPU and system architecture design. Things that appear to be simple, like figuring out the instruction set architecture, can actually be fairly difficult when you sit down and try to figure out the best way to do it. Fortunately with my project I was able to take a simpler approach for some of these problems as the whole goal of this project was simplicity. On a *slightly* less technical level I feel that I improved my overall ability to gage the complexity of projects of this nature and improved my approach to solving them.

There are literally an infinite number of things that could be improved or added to this project. Anything from new CPU instructions, to peripherals, to compiler directives. A few features that I feel would be interesting to add are peripherals for I2C and SPI (common communication protocols), a small VGA peripheral that exposes one side of a dual port ram to the CPU, an example program that can read and write to an SD card (which uses SPI), addition math instructions such as divide and multiply, compiler directives to include arbitrary data in the ROM such as text and images, a directive to specify where certain blocks of instructions are loaded into RAM, basic interrupt support... the possibilities for expansion are endless.

All project files can be viewed here:
`https://github.com/bear24rw/EECE9060.git`

Or as a `.zip` download:
`https://github.com/bear24rw/EECE9060/archive/master.zip`

References

[1] http://colinmackenzie.net/index.php?option=com_content&view=article&id=11:your-first-cpu&catid=8:rotator&Itemid=7

[2] http://aquamentus.com/flex_bison.html

[3] http://en.wikipedia.org/wiki/Instruction_cycle

[4] http://en.wikipedia.org/wiki/Von_Neumann_architecture

[5] http://en.wikipedia.org/wiki/Self-modifying_code

[6] http://en.wikipedia.org/wiki/Harvard_architecture

[7] http://en.wikipedia.org/wiki/Memory-mapped_I/O

[8] http://quartushelp.altera.com/13.0/mergedProjects/reference/glossary/def_mif.htm

Example Programs

```
1  .define LEDR GPO_0  ; red leds are on GPO 0
2  .define LEDG GPO_1  ; green leds are on GPO 1
3
4  ldl r0,0            ; initialize register 0 to 0
5
6  main:
7      st r0,LEDG      ; output register 0 to green leds
8      st r0,LEDR      ; output register 0 to red leds
9      inc r0          ; increment register 0
10
11     stl 0,TMR_RST   ; reset the timer by writing to the TMR_RST address
12
13 delay:
14     ld r1,TMR_TRIG  ; get the status of the timer trigger
15     brz r1,delay    ; if its still zero keep delaying
16
17     jmp main        ; go back and do it all again
```

Listing 2: count.asm

```
1  .define LEDR GPO_0
2  .define LEDG GPO_1
3
4  ; clear all leds
5  stl 0,LEDR
6  stl 0,LEDG
7
8  ldl r0,1      ; green led register
9  ldl r1,0      ; red led register
10 ldl r4,1      ; shift amount
11
12 green_left:
13
14     stl 0,TMR_RST        ; reset the timer
15 delay_0:                 ; wait for timer to trigger
16     ld r5,TMR_TRIG
17     brz r5,delay_0
18
19     st r0,LEDG           ; output R0 to green leds
20     sfl r0,r0,r4         ; shift R0 to left by R4 (R4=1)
21     brnz r0,green_left   ; keep shifting until we roll of left side
22
23 ; green rolled off left side
24 ldl r1,1                 ; set R1 to turn on right most led
25 st r0,LEDG               ; output R0 to green leds
26 st r1,LEDR               ; output R1 to red leds
27
28 red_left:
29
30     stl 0,TMR_RST        ; reset the timer
31 delay_1:                 ; wait for timer to trigger
32     ld r5,TMR_TRIG
33     brz r5,delay_1
34
35     st r1,LEDR           ; output R1 to red leds
36     sfl r1,r1,r4         ; shift R1 to left by R4 (R4=1)
37     brnz r1,red_left     ; if leds are zero we went off left side
38
39 ; red rolled off left side
40 ldl r0,1                 ; set R0 to turn on right most led
41 st r1,LEDR               ; output R1 to red leds
42 st r0,LEDG               ; output R0 to green leds
43
44 jmp green_left
```

Listing 3: shift.asm

```
1  .define LEDR GPO_0
2  .define LEDG GPO_1
3
4  ; 1Hz blink = 0.5s delays
5  ; 50Mhz clock = 20ns
6  ; 0.5s / 20ns = 2.5E7 = 01 7D 78 40
7  stl 0x01,TMR_3
8  stl 0x7D,TMR_2
9  stl 0x78,TMR_1
10 stl 0x40,TMR_0
11
12 ldl r0,0xFF          ; led value will be stored in R0
13
14 loop:
15
16     inv r0,r0        ; flip state of R0 (R0 = ~R0)
17     st r0,LEDG       ; update the leds
18
19     stl 0,TMR_RST    ; reset the timer
20 delay:               ; wait for timer to trigger
21     ld r1,TMR_TRIG
22     brz r1,delay
23
24     jmp loop
```

Listing 4: blink_1hz.asm

```
.define HEX_0 GPO_2
.define HEX_1 GPO_3
.define HEX_2 GPO_4
.define HEX_3 GPO_5

ldl r0, 0    ; 1s place
ldl r1, 0    ; 10s place
ldl r2, 0    ; 100s place
ldl r3, 0    ; 1000s place

ldl r4, 10

loop:

    inc r0                  ; increment the 1s place
    eql r5,r0,r4            ; check if it is now == 10
    brz r5,display          ; if nots 10 display current number
    ldl r0,0                ; it was 10 so reset it to 0

    inc r1                  ; increment the 10s place
    eql r5,r1,r4            ; check if it is now == 10
    brz r5,display          ; if not 10 display current number
    ldl r1,0                ; it was 10 so reset it to 0

    inc r2                  ; increment the 100s place
    eql r5,r2,r4            ; check if it is now == 10
    brz r5,display          ; if not 10 display current number
    ldl r2,0                ; it was 10 so reset it to 0

    inc r3                  ; increment the 1000s place
    eql r5,r3,r4            ; check if it is now == 10
    brz r5,display          ; if not 10 display current number
    ldl r3,0                ; it was 10 so reset it to 0

display:                    ; update the HEX displays
    st r0,HEX_0
    st r1,HEX_1
    st r2,HEX_2
    st r3,HEX_3

    stl 0,TMR_RST           ; reset the timer
delay_loop:                 ; wait for timer to trigger
    ld r10,TMR_TRIG
    brz r10,delay_loop

    jmp loop
```

Listing 5: seg_count.asm

Python Code

```
#!/usr/bin/env python

import sys
import constants
import re

def bits(number):
    return bin(number)[2:].zfill(8)

def h_byte(x):
    return x >> 8

def l_byte (x):
    return x & 0x00FF

if __name__ == "__main__":

    asm_filename = sys.argv[1]
    rom_filename = sys.argv[1].replace("asm", "rom")
    txt_filename = sys.argv[1].replace("asm", "txt")

    asm_file = open(asm_filename)
    rom_file = open(rom_filename, 'wb')
    txt_file = open(txt_filename, 'w')

    bytes = []

    defines = {}
    labels = {}
    jumps = []

    #
    # Pad up until the reset vector
    #
    for _ in range(constants.reset_vector):
        bytes.append(0)

    #
    # Fill out instruction bytes
    #
    for line in asm_file:

        line = line.upper().strip()

        if ';' in line:
            line = line[:line.find(';')].strip()

        if len(line) == 0: continue

        if line.startswith(".DEFINE"):
            _, original, new = line.split()
            defines[original] = new
            continue

        for word in defines:
            line = line.replace(word, defines[word])

        if ":" in line:
            name = line.replace(':','')
            if name in labels:
                print "Duplicate label: " + name
                sys.exit(1)
            labels[name] = len(bytes)
            continue

        if line == 'HALT':
            bytes.append(constants.op_codes[line])
            bytes.append(0)
            bytes.append(0)
            bytes.append(0)
            continue

        op, args = line.split(' ', 1)

        bytes.append(constants.op_codes[op])

        if op in ('JMP'):
            jumps.append({'addr': len(bytes), 'label': args})
            bytes.append(0)
            bytes.append(0)
            bytes.append(0)
            continue

        args = [x.strip() for x in args.split(",")]

        # remove the R off the register names (R5 -> 5)
        args = [re.sub(r'R(\d+)', r'\1', x) for x in args]

        for i, arg in enumerate(args):
            if arg in constants.address_names:
                args[i] = constants.address_names[arg]

        if op in ('BRZ', 'BRNZ'):
            jumps.append({'addr': len(bytes)+1, 'label': args[1]})
            bytes.append(int(args[0]))
            bytes.append(0)
            bytes.append(0)
            continue

        # convert strings of decimal, hex, or binary to ints
        for i, arg in enumerate(args):
            if not isinstance(arg, int):
                args[i] = eval(arg)

```

```
105             if op in ('LD', 'ST', 'STL'):
106                 d, addr = args
107                 bytes.append(d)
108                 bytes.append(h_byte(addr))
109                 bytes.append(l_byte(addr))
110                 continue
111
112             if op in ('LDL'):
113                 d, value = args
114                 bytes.append(d)
115                 bytes.append(value)
116                 bytes.append(0)
117                 continue
118
119             if op in ('MOV', 'INV'):
120                 d, a = args
121                 bytes.append(d)
122                 bytes.append(a)
123                 bytes.append(0)
124                 continue
125
126             if op in ('INC', 'DEC'):
127                 bytes.append(args[0])
128                 bytes.append(0)
129                 bytes.append(0)
130                 continue
131
132             d, a, b = args
133             bytes.append(d)
134             bytes.append(a)
135             bytes.append(b)
136
137         #
138         # Pad the rest of the rom with 0s
139         #
140         for _ in range(constants.ram_size - len(bytes)):
141             bytes.append(0)
142
143         #
144         # Go through all the jump instructions and fill out the address
145         #
146         for jump in jumps:
147             bytes[jump['addr']+0] = h_byte(labels[jump['label']])
148             bytes[jump['addr']+1] = l_byte(labels[jump['label']])
149
150         #
151         # Write all bytes out
152         #
153         for byte in bytes:
154             txt_file.write(bits(byte)+'\n')
155             rom_file.write(chr(byte))
```

Listing 6: asm.py

```
1   #!/usr/bin/env python
2
3   import os
4   import sys
5   import serial
6   import time
7
8   s = serial.Serial('/dev/ttyS0', 115200)
9
10  rom_file = sys.argv[1]
11  rom_size = os.path.getsize(rom_file)
12
13  print "Rom size: %d" % rom_size
14
15  rom = open(sys.argv[1], 'rb')
16
17  bytes_sent = 0
18  percent_sent = 0.0
19
20  print "Waiting for FPGA to request bytes..."
21
22  for sent_byte in rom.read():
23
24      s.write(sent_byte)
25
26      recv_byte = s.read()
27
28      bytes_sent += 1
29      percent_sent = float(bytes_sent)/float(rom_size)*100.0
30
31      # we should receive back the last byte we sent as an ACK
32      if (recv_byte != sent_byte):
33          print "RECIEVED BYTE DOES NOT MATCH!"
34          print "Recv: %2X | Sent: %2X" % (ord(recv_byte), ord(sent_byte))
35          sys.exit()
36
37      print "[%.2f] %d / %d (sent: %2X | recv: %2X)" % \
38              (percent_sent, bytes_sent, rom_size, ord(sent_byte), ord(recv_byte) )
```

Listing 7: send_rom.py

Verilog Code

```
module top(
    input CLOCK_50,

    input [9:0] SW,
    input [3:0] KEY,

    output [9:0] LEDR,
    output [7:0] LEDG,

    output [6:0] HEX0,
    output [6:0] HEX1,
    output [6:0] HEX2,
    output [6:0] HEX3,

    input  UART_RXD,
    output UART_TXD
);

    wire [15:0] ram_addr;
    wire [7:0]  ram_di;
    wire [7:0]  ram_do;
    wire        ram_we;

    wire [7:0] hex_0;
    wire [7:0] hex_1;
    wire [7:0] hex_2;
    wire [7:0] hex_3;

    soc soc(
        .clk(CLOCK_50),
        .boot_trigger(~KEY[0]),
        .booting(LEDR[9]),

        .ram_addr(ram_addr),
        .ram_di(ram_di),
        .ram_do(ram_do),
        .ram_we(ram_we),

        .gpi_0(SW[7:0]),
        .gpi_1({4'b0, KEY}),
        .gpi_2(),
        .gpi_3(),
        .gpi_4(),
        .gpi_5(),
        .gpi_6(),
        .gpi_7(),

        .gpo_0(LEDR[7:0]),
        .gpo_1(LEDG[7:0]),
        .gpo_2(hex_0),
        .gpo_3(hex_1),
        .gpo_4(hex_2),
        .gpo_5(hex_3),
        .gpo_6(),
        .gpo_7(),

        .uart_rxd(UART_RXD),
        .uart_txd(UART_TXD)
    );

    ram ram(
        .clk(CLOCK_50),
        .addr(ram_addr[`RAM_ADDR_BITS-1:0]),
        .we(ram_we),
        .do(ram_do),
        .di(ram_di)
    );

    seven_seg s0(hex_0, HEX0);
    seven_seg s1(hex_1, HEX1);
    seven_seg s2(hex_2, HEX2);
    seven_seg s3(hex_3, HEX3);

endmodule
```

Listing 8: top.v

```
module soc(
    input clk,

    input  boot_trigger,
    output booting,

    output [15:0] ram_addr,
    output [7:0]  ram_di,
    input  [7:0]  ram_do,
    output        ram_we,

    input [7:0] gpi_0,
    input [7:0] gpi_1,
    input [7:0] gpi_2,
    input [7:0] gpi_3,
    input [7:0] gpi_4,
    input [7:0] gpi_5,
    input [7:0] gpi_6,
    input [7:0] gpi_7,

    output [7:0] gpo_0,
    output [7:0] gpo_1,
    output [7:0] gpo_2,
    output [7:0] gpo_3,
    output [7:0] gpo_4,
    output [7:0] gpo_5,
    output [7:0] gpo_6,
    output [7:0] gpo_7,

    input  uart_rxd,
    output uart_txd
);

    // ------------------------------------------------------
    //                          CPU
    // ------------------------------------------------------

    wire cpu_clk = ~clk;
    wire [15:0] cpu_addr;
    wire [7:0]  cpu_di;
    wire [7:0]  cpu_do;
    wire        cpu_we;
    wire        cpu_rst;

    cpu cpu(
        .clk(cpu_clk),
        .rst(cpu_rst),
        .addr(cpu_addr),
        .di(cpu_di),
        .do(cpu_do),
        .we(cpu_we)
    );

    // ------------------------------------------------------
    //                    RAM INTERFACE
    // ------------------------------------------------------

    assign ram_addr = booting ? boot_addr : cpu_addr;
    assign ram_di   = booting ? boot_data : cpu_do;
    assign ram_we   = booting ? 1         : cpu_we;

    // ------------------------------------------------------
    //                    MEMORY MAPPER
    // ------------------------------------------------------

    mem_mapper mem_mapper(
        .addr(cpu_addr),
        .cpu_di(cpu_di),
        .ram_do(ram_do),
        .gpi_0(gpi_0),
        .gpi_1(gpi_1),
        .gpi_2(gpi_2),
        .gpi_3(gpi_3),
        .gpi_4(gpi_4),
        .gpi_5(gpi_5),
        .gpi_6(gpi_6),
        .gpi_7(gpi_7),
        .timer_do(timer_do)
    );

    // ------------------------------------------------------
    //                          TIMERS
    // ------------------------------------------------------

    wire [7:0] timer_do;

    timer timer(
        .clk(clk),
        .rst(cpu_rst),
        .addr(cpu_addr),
        .we(cpu_we),
        .do(timer_do),
        .di(cpu_do)
    );

    // ------------------------------------------------------
    //                GENERAL PURPOSE OUTPUTS
    // ------------------------------------------------------

    gpo #(.addr(`ADDR_GPO_0)) gpo0(clk, cpu_addr, cpu_do, cpu_we, gpo_0);
    gpo #(.addr(`ADDR_GPO_1)) gpo1(clk, cpu_addr, cpu_do, cpu_we, gpo_1);
    gpo #(.addr(`ADDR_GPO_2)) gpo2(clk, cpu_addr, cpu_do, cpu_we, gpo_2);
    gpo #(.addr(`ADDR_GPO_3)) gpo3(clk, cpu_addr, cpu_do, cpu_we, gpo_3);
    gpo #(.addr(`ADDR_GPO_4)) gpo4(clk, cpu_addr, cpu_do, cpu_we, gpo_4);
    gpo #(.addr(`ADDR_GPO_5)) gpo5(clk, cpu_addr, cpu_do, cpu_we, gpo_5);
    gpo #(.addr(`ADDR_GPO_6)) gpo6(clk, cpu_addr, cpu_do, cpu_we, gpo_6);
    gpo #(.addr(`ADDR_GPO_7)) gpo7(clk, cpu_addr, cpu_do, cpu_we, gpo_7);
```

```
108
109         // ------------------------------------------------------
110         //                          UART
111         // ------------------------------------------------------
112
113         wire        uart_rst      = booting ? boot_rst      : cpu_rst;
114         wire        uart_transmit = booting ? boot_transmit : 'b0;
115         wire [7:0]  uart_tx_data  = booting ? boot_tx_data  : 'b0;
116         wire [7:0]  uart_rx_data;
117         wire        uart_rx_done;
118         wire        uart_tx_done;
119
120         uart uart(
121             .sys_clk(clk),
122             .sys_rst(uart_rst),
123             .uart_rx(uart_rxd),
124             .uart_tx(uart_txd),
125             .divisor(50000000/115200/16),
126             .rx_data(uart_rx_data),
127             .tx_data(uart_tx_data),
128             .rx_done(uart_rx_done),
129             .tx_done(uart_tx_done),
130             .tx_wr(uart_transmit)
131         );
132
133         // ------------------------------------------------------
134         //                       BOOTLOADER
135         // ------------------------------------------------------
136
137         wire boot_rst;
138         wire [15:0] boot_addr;
139         wire [7:0]  boot_data;
140         wire [7:0]  boot_tx_data;
141         wire        boot_transmit;
142
143         bootloader bootloader(
144             .clk(clk),
145             .rx_data(uart_rx_data),
146             .tx_data(boot_tx_data),
147             .rx_done(uart_rx_done),
148             .tx_done(uart_tx_done),
149             .transmit(boot_transmit),
150             .ram_addr(boot_addr),
151             .ram_data(boot_data),
152             .trigger(boot_trigger),
153             .booting(booting),
154             .cpu_rst(cpu_rst),
155             .boot_rst(boot_rst)
156         );
157
158     endmodule
```

Listing 9: soc.v

```
1   // http://www.altera.com/support/examples/verilog/ver-single-port-ram.html
2
3   module ram(
4       input clk,
5       input we,
6       input [ADDR_BITS-1:0] addr,
7       input [7:0] di,
8       output [7:0] do
9   );
10
11      parameter WIDTH     = 8;                    // 8 bits wide
12      parameter ADDR_BITS = `RAM_ADDR_BITS;       // 2**13 (8KB) deep
13
14      reg [WIDTH-1:0] ram[(2**ADDR_BITS)-1:0];
15
16      reg [ADDR_BITS-1:0] addr_reg = 0;
17
18      always @(posedge clk) begin
19
20          // if write enable store new value
21          if (we) ram[addr] <= di;
22
23          // save this addr so we can continue to output it
24          addr_reg <= addr;
25
26      end
27
28      // continuous assignment implies read returns NEW data
29      // this is the natural behavior of the TriMatrix memory
30      // blocks in single port mode
31      assign do = ram[addr_reg];
32
33  endmodule
```

Listing 10: ram.v

```
`include "constants.v"

module cpu(
    input             clk,
    input             rst,
    output    [15:0] addr,
    input      [7:0] di,
    output reg [7:0] do,
    output            we
);

    // ---------------------------------
    // Instruction Cycle State Machine
    // ---------------------------------

    parameter FETCH_0 = 0;
    parameter FETCH_1 = 1;
    parameter FETCH_2 = 2;
    parameter FETCH_3 = 3;
    parameter DECODE  = 4;
    parameter EXECUTE = 5;
    parameter STORE   = 6;

    reg [7:0] state = FETCH_0;

    always @(posedge clk, posedge rst) begin
        if (rst) begin
            state <= STORE;
        end else begin
            case (state)
                FETCH_0: state <= FETCH_1;
                FETCH_1: state <= FETCH_2;
                FETCH_2: state <= FETCH_3;
                FETCH_3: state <= DECODE;
                DECODE:  state <= EXECUTE;
                EXECUTE: state <= STORE;
                STORE:   state <= FETCH_0;
            endcase
        end
    end


    // ---------------------------------
    //       Internal registers
    // ---------------------------------

    reg [15:0] PC = `RESET_VECTOR;
    reg [31:0] IR = 'b0;
    reg [7:0] regs[0:255];
    reg [7:0] w_reg;

    wire [7:0]  op_code = IR[31:24];
    wire [7:0]  op_d    = IR[23:16];
    wire [7:0]  op_a    = IR[15:8];
    wire [7:0]  op_b    = IR[7:0];
    wire [15:0] d_addr  = IR[15:0];
    wire [15:0] jmp_addr = IR[23:8];
    wire [15:0] br_addr = IR[15:0];

    reg [15:0] i_addr = 'b0;
    reg get_data = 'b0;

    assign addr = get_data ? d_addr : i_addr;
    assign we = (op_code == `OP_ST || op_code == `OP_STL) && (state == STORE);

    // ---------------------------------
    // Instruction address state machine
    // ---------------------------------

    always @(posedge clk, posedge rst) begin
        if (rst) begin
            i_addr <= `RESET_VECTOR;
        end else begin
            case (state)
                STORE:   i_addr <= PC + 0;
                FETCH_0: i_addr <= PC + 1;
                FETCH_1: i_addr <= PC + 2;
                FETCH_2: i_addr <= PC + 3;
            endcase
        end
    end

    // ---------------------------------
    //       Execution state machine
    // ---------------------------------

    always @(posedge clk, posedge rst) begin
        if (rst) begin
            PC <= `RESET_VECTOR;
            IR <= 0;
            do <= 0;
            get_data <= 0;
        end else begin
            case (state)

                FETCH_0: IR[31:24] <= di;
                FETCH_1: IR[23:16] <= di;
                FETCH_2: IR[15:8]  <= di;
                FETCH_3: IR[7:0]   <= di;

                DECODE: begin
                    get_data <= 1;
                    if (op_code != `OP_HALT) begin
                        PC <= PC + 4;
                    end
                end
```

```
108                     EXECUTE: begin
109                         case (op_code)
110                             `OP_ST:   do <= regs[op_d];
111                             `OP_STL:  do <= op_d;
112                             `OP_LD:   regs[op_d] <= di;
113                             `OP_LDL:  regs[op_d] <= op_a;
114                             `OP_MOV:  regs[op_d] <= regs[op_a];
115
116                             `OP_ADD:  regs[op_d] <= regs[op_a] + regs[op_b];
117                             `OP_SUB:  regs[op_d] <= regs[op_a] - regs[op_b];
118                             `OP_AND:  regs[op_d] <= regs[op_a] & regs[op_b];
119                             `OP_OR:   regs[op_d] <= regs[op_a] | regs[op_b];
120                             `OP_XOR:  regs[op_d] <= regs[op_a] ^ regs[op_b];
121                             `OP_SFL:  regs[op_d] <= regs[op_a] << regs[op_b];
122                             `OP_SFR:  regs[op_d] <= regs[op_a] >> regs[op_b];
123                             `OP_INC:  regs[op_d] <= regs[op_d] + 1;
124                             `OP_DEC:  regs[op_d] <= regs[op_d] - 1;
125                             `OP_EQL:  regs[op_d] <= regs[op_a] == regs[op_b];
126                             `OP_GTH:  regs[op_d] <= regs[op_a] > regs[op_b];
127                             `OP_LTH:  regs[op_d] <= regs[op_a] < regs[op_b];
128                             `OP_INV:  regs[op_d] <= ~regs[op_a];
129
130                             `OP_BRZ:  if (regs[op_d] == 0) PC <= br_addr;
131                             `OP_BRNZ: if (regs[op_d] != 0) PC <= br_addr;
132
133                             `OP_JMP:  PC <= jmp_addr;
134                         endcase
135
136                     end
137
138                     STORE: begin
139                         get_data <= 0;
140                     end
141                 endcase
142             end
143         end
144
145         // ----------------------------------
146         //       Simulation Debug Message
147         // ----------------------------------
148
149         /*
150         always @(posedge clk, posedge rst) begin
151             if (rst) begin
152                 $display("[cpu] In reset");
153             end else begin
154                 if (state == DECODE) begin
155                     case (op_code)
156                         `OP_HALT: $display("[cpu] [decode] PC: %d IR: %x op_code: HALT" , PC, IR);
157                         `OP_LD:   $display("[cpu] [decode] PC: %d IR: %x op_code: LD (r[%x] = M[%x]", PC, IR, op_d, d_addr);
158                         `OP_ST:   $display("[cpu] [decode] PC: %d IR: %x op_code: ST (%d = r[%d])", PC, IR, d_addr, op_d);
159                         `OP_STL:  $display("[cpu] [decode] PC: %d IR: %x op_code: STL (%d = %d)", PC, IR, d_addr, op_d);
160                         `OP_LDL:  $display("[cpu] [decode] PC: %d IR: %x op_code: LDL (r[%d] = %d", PC, IR, op_d, op_a);
161                         `OP_MOV:  $display("[cpu] [decode] PC: %d IR: %x op_code: MOV"  , PC, IR);
162                         `OP_ADD:  $display("[cpu] [decode] PC: %d IR: %x op_code: ADD (r[%d]: %d + r[%d]: %d"
    , PC, IR, op_a, regs[op_a], op_b, regs[op_b]);
163                         `OP_SUB:  $display("[cpu] [decode] PC: %d IR: %x op_code: SUB"  , PC, IR);
164                         `OP_AND:  $display("[cpu] [decode] PC: %d IR: %x op_code: AND"  , PC, IR);
165                         `OP_OR:   $display("[cpu] [decode] PC: %d IR: %x op_code: OR"   , PC, IR);
166                         `OP_XOR:  $display("[cpu] [decode] PC: %d IR: %x op_code: XOR"  , PC, IR);
167                         `OP_SFL:  $display("[cpu] [decode] PC: %d IR: %x op_code: SFL" , PC, IR);
168                         `OP_SFR:  $display("[cpu] [decode] PC: %d IR: %x op_code: SFR" , PC, IR);
169                         `OP_INC:  $display("[cpu] [decode] PC: %d IR: %x op_code: INC", PC, IR);
170                         `OP_DEC:  $display("[cpu] [decode] PC: %d IR: %x op_code: DEC", PC, IR);
171                         `OP_EQL:  $display("[cpu] [decode] PC: %d IR: %x op_code: EQL", PC, IR);
172                         `OP_GTH:  $display("[cpu] [decode] PC: %d IR: %x op_code: GTH", PC, IR);
173                         `OP_LTH:  $display("[cpu] [decode] PC: %d IR: %x op_code: LTH", PC, IR);
174                         `OP_BRZ:  $display("[cpu] [decode] PC: %d IR: %x op_code: BRZ", PC, IR);
175                         `OP_BRNZ: $display("[cpu] [decode] PC: %d IR: %x op_code: BRNZ", PC, IR);
176                         `OP_JMP:  $display("[cpu] [decode] PC: %d IR: %x op_code: JMP"  , PC, IR);
177                         default:  $display("[cpu] [decode] ERROR: Invalid op_code: %b (%d) IR: %b", op_code, op_code, IR);
178                     endcase
179                 end
180             end
181         end
182
183         always @(posedge clk) begin
184             if (state == EXECUTE) begin
185                 case (op_code)
186                     `OP_LD: $display("[cpu] [exec] regs[%x] = %x", op_d, di);
187                     `OP_JMP: $display("[cpu] [exec] jumping to: %x", jmp_addr);
188                 endcase
189             end
190         end
191
192         always @(posedge clk) begin
193             if (rst == 0) begin
194                 case (state)
195                     FETCH_0: $display("[cpu] [F0] PC: %d i_addr: %d di: %x", PC, i_addr, di);
196                     FETCH_1: $display("[cpu] [F1] PC: %d i_addr: %d di: %x", PC, i_addr, di);
197                     FETCH_2: $display("[cpu] [F2] PC: %d i_addr: %d di: %x", PC, i_addr, di);
198                     FETCH_3: $display("[cpu] [F3] PC: %d i_addr: %d di: %x", PC, i_addr, di);
199                 endcase
200             end
201         end
202         */
203
204 endmodule
```

Listing 11: cpu.v

```
`include "constants.v"

module bootloader(
    input clk,

    input       [7:0]  rx_data,
    output reg  [7:0]  tx_data,
    input              rx_done,
    input              tx_done,
    output reg         transmit,

    output reg  [15:0] ram_addr,
    output reg   [7:0] ram_data,

    input              trigger,
    output reg         booting,
    output reg         cpu_rst,
    output reg         boot_rst
);

    initial booting = 1;
    initial cpu_rst = 0;
    initial boot_rst = 0;

    // ------------------------------------------------------
    //                ROM LOADING STATE MACHINE
    // ------------------------------------------------------

    localparam S_BOOT_RST_H     = 0;    // pull boot reset flag high
    localparam S_BOOT_RST_L     = 1;    // pull boot reset flag low
    localparam S_RECV           = 2;    // wait for data byte
    localparam S_SEND           = 3;    // send that back back to ACK
    localparam S_WRITE          = 4;    // write data to RAM
    localparam S_CPU_RST_H      = 5;    // pull cpu reset high
    localparam S_CPU_RST_L      = 6;    // pull cpu reset low
    localparam S_DONE           = 7;    // idle

    reg [3:0] state = S_DONE;

    always @(posedge clk) begin
        if (trigger) begin
            boot_rst <= 0;
            booting <= 1;
            cpu_rst <= 0;
            ram_addr <= 0;
            state <= S_BOOT_RST_H;
            transmit <= 0;
            tx_data <= 0;
        end else begin
            case (state)

                S_BOOT_RST_H: begin
                    boot_rst <= 1;
                    state <= S_BOOT_RST_L;
                end

                S_BOOT_RST_L: begin
                    boot_rst <= 0;
                    state <= S_RECV;
                end

                S_RECV: begin
                    if (rx_done) begin
                        tx_data <= rx_data;
                        ram_data <= rx_data;
                        transmit <= 1;
                        state <= S_SEND;
                    end
                end

                S_SEND: begin
                    transmit <= 0;
                    if (tx_done) begin
                        state <= S_WRITE;
                    end
                end

                S_WRITE: begin
                    if (ram_addr == (2**`RAM_ADDR_BITS)-1) begin
                        state <= S_CPU_RST_H;
                    end else begin
                        ram_addr <= ram_addr + 1;
                        state <= S_RECV;
                    end
                end

                S_CPU_RST_H: begin
                    cpu_rst <= 1;
                    booting <= 0;
                    state <= S_CPU_RST_L;
                end

                S_CPU_RST_L: begin
                    cpu_rst <= 0;
                end

            endcase
        end
    end

endmodule
```

Listing 12: bootloader.v

```
1  `include "constants.v"
2
3  module mem_mapper(
4      input   [15:0]  addr,
5      output  [7:0]   cpu_di,
6
7      input   [7:0]   gpi_0,
8      input   [7:0]   gpi_1,
9      input   [7:0]   gpi_2,
10     input   [7:0]   gpi_3,
11     input   [7:0]   gpi_4,
12     input   [7:0]   gpi_5,
13     input   [7:0]   gpi_6,
14     input   [7:0]   gpi_7,
15     input   [7:0]   timer_do,
16     input   [7:0]   ram_do
17 );
18
19     assign cpu_di = (addr == `ADDR_GPI_0)     ? gpi_0     :
20                     (addr == `ADDR_GPI_1)     ? gpi_1     :
21                     (addr == `ADDR_GPI_2)     ? gpi_2     :
22                     (addr == `ADDR_GPI_3)     ? gpi_3     :
23                     (addr == `ADDR_GPI_4)     ? gpi_4     :
24                     (addr == `ADDR_GPI_5)     ? gpi_5     :
25                     (addr == `ADDR_GPI_6)     ? gpi_6     :
26                     (addr == `ADDR_GPI_7)     ? gpi_7     :
27                     (addr == `ADDR_TMR_TRIG) ? timer_do :
28                     ram_do;
29
30 endmodule
```

Listing 13: mem_mapper.v

```
1  module gpo(
2      input clk,
3      input [15:0] cpu_addr,
4      input [7:0]  cpu_do,
5      input        cpu_we,
6      output reg [7:0] do
7  );
8
9      initial do = 0;
10
11     parameter addr = 0;
12
13     always @(posedge clk) begin
14         if (cpu_we && (cpu_addr == addr)) begin
15             do <= cpu_do;
16         end
17     end
18
19 endmodule
```

Listing 14: gpo.v

```
1  `include "constants.v"
2
3  module timer(
4      input clk,
5      input rst,
6
7      input [15:0] addr,
8      input we,
9      input [7:0] di,
10     output [7:0] do
11 );
12
13     // -----------------------------------------------------
14     //                      TRIGGER LEVEL
15     // -----------------------------------------------------
16
17     // default the timer to trigger every 100ms
18     // 50Mhz clock = 20ns
19     // 100ms / 20ns = 5*10^6 = 00 4C 4B 40
20
21     `ifndef SIMULATION
22         localparam DEFAULT_3 = 8'h00;
23         localparam DEFAULT_2 = 8'h4C;
24         localparam DEFAULT_1 = 8'h4B;
25         localparam DEFAULT_0 = 8'h40;
26     `else
27         localparam DEFAULT_3 = 8'h00;
28         localparam DEFAULT_2 = 8'h00;
29         localparam DEFAULT_1 = 8'h00;
30         localparam DEFAULT_0 = 8'h40;
31     `endif
32
33     reg [7:0] byte_3 = DEFAULT_3;
34     reg [7:0] byte_2 = DEFAULT_2;
35     reg [7:0] byte_1 = DEFAULT_1;
36     reg [7:0] byte_0 = DEFAULT_0;
37
38     wire [31:0] trigger_value = {byte_3, byte_2, byte_1, byte_0};
39
40     always @(posedge clk, posedge rst) begin
41         if (rst) begin
42             byte_3 <= DEFAULT_3;
43             byte_2 <= DEFAULT_2;
44             byte_1 <= DEFAULT_1;
45             byte_0 <= DEFAULT_0;
46         end else begin
47             if (we && (addr == `ADDR_TMR_3)) begin byte_3 <= di; end
48             if (we && (addr == `ADDR_TMR_2)) begin byte_2 <= di; end
49             if (we && (addr == `ADDR_TMR_1)) begin byte_1 <= di; end
50             if (we && (addr == `ADDR_TMR_0)) begin byte_0 <= di; end
51         end
52     end
53
54     // -----------------------------------------------------
55     //                         COUNTER
56     // -----------------------------------------------------
57
58     reg [31:0] count = 0;
59     reg triggered = 0;
60     assign do = {7'b0, triggered};
61
62     always @(posedge clk, posedge rst) begin
63         if (rst) begin
64             count <= 0;
65             triggered <= 0;
66         end else begin
67             if (count == trigger_value) begin
68                 triggered <= 1;
69                 count <= 0;
70             end else if (we && (addr == `ADDR_TMR_RST)) begin
71                 count <= 0;
72                 triggered <= 0;
73             end else begin
74                 count <= count + 1;
75             end
76         end
77     end
78
79 endmodule
```

Listing 15: timer.v

```
/*
 * Milkymist VJ SoC
 * Copyright (C) 2007, 2008, 2009, 2010 Sebastien Bourdeauducq
 * Copyright (C) 2007 Das Labor
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, version 3 of the License.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */

module uart(
	input sys_clk,
	input sys_rst,

	input uart_rx,
	output reg uart_tx,

	input [15:0] divisor,

	output reg [7:0] rx_data,
	output reg rx_done,

	input [7:0] tx_data,
	input tx_wr,
	output reg tx_done
);

//--------------------------------------------------------------------------
// enable16 generator
//--------------------------------------------------------------------------
reg [15:0] enable16_counter;

wire enable16;
assign enable16 = (enable16_counter == 16'd0);

always @(posedge sys_clk) begin
	if(sys_rst)
		enable16_counter <= divisor - 16'b1;
	else begin
		enable16_counter <= enable16_counter - 16'd1;
		if(enable16)
			enable16_counter <= divisor - 16'b1;
	end
end

//--------------------------------------------------------------------------
// Synchronize uart_rx
//--------------------------------------------------------------------------
reg uart_rx1;
reg uart_rx2;

always @(posedge sys_clk) begin
	uart_rx1 <= uart_rx;
	uart_rx2 <= uart_rx1;
end

//--------------------------------------------------------------------------
// UART RX Logic
//--------------------------------------------------------------------------
reg rx_busy;
reg [3:0] rx_count16;
reg [3:0] rx_bitcount;
reg [7:0] rx_reg;

always @(posedge sys_clk) begin
	if(sys_rst) begin
		rx_done <= 1'b0;
		rx_busy <= 1'b0;
		rx_count16  <= 4'd0;
		rx_bitcount <= 4'd0;
	end else begin
		rx_done <= 1'b0;

		if(enable16) begin
			if(~rx_busy) begin // look for start bit
				if(~uart_rx2) begin // start bit found
					rx_busy <= 1'b1;
					rx_count16 <= 4'd7;
					rx_bitcount <= 4'd0;
				end
			end else begin
				rx_count16 <= rx_count16 + 4'd1;

				if(rx_count16 == 4'd0) begin // sample
					rx_bitcount <= rx_bitcount + 4'd1;

					if(rx_bitcount == 4'd0) begin // verify startbit
						if(uart_rx2)
							rx_busy <= 1'b0;
					end else if(rx_bitcount == 4'd9) begin
						rx_busy <= 1'b0;
						if(uart_rx2) begin // stop bit ok
							rx_data <= rx_reg;
							rx_done <= 1'b1;
						end // ignore RX error
					end else
						rx_reg <= {uart_rx2, rx_reg[7:1]};
				end
			end
```

```
108                     end
109             end
110     end
111
112     //------------------------------------------------------------------
113     // UART TX Logic
114     //------------------------------------------------------------------
115     reg tx_busy;
116     reg [3:0] tx_bitcount;
117     reg [3:0] tx_count16;
118     reg [7:0] tx_reg;
119
120     always @(posedge sys_clk) begin
121             if(sys_rst) begin
122                     tx_done <= 1'b0;
123                     tx_busy <= 1'b0;
124                     uart_tx <= 1'b1;
125             end else begin
126                     tx_done <= 1'b0;
127                     if(tx_wr) begin
128                             tx_reg <= tx_data;
129                             tx_bitcount <= 4'd0;
130                             tx_count16 <= 4'd1;
131                             tx_busy <= 1'b1;
132                             uart_tx <= 1'b0;
133     `ifdef SIMULATION
134                             $display("UART:  %c", tx_data);
135     `endif
136                     end else if(enable16 && tx_busy) begin
137                             tx_count16  <= tx_count16 + 4'd1;
138
139                             if(tx_count16 == 4'd0) begin
140                                     tx_bitcount <= tx_bitcount + 4'd1;
141
142                                     if(tx_bitcount == 4'd8) begin
143                                             uart_tx <= 1'b1;
144                                     end else if(tx_bitcount == 4'd9) begin
145                                             uart_tx <= 1'b1;
146                                             tx_busy <= 1'b0;
147                                             tx_done <= 1'b1;
148                                     end else begin
149                                             uart_tx <= tx_reg[0];
150                                             tx_reg <= {1'b0, tx_reg[7:1]};
151                                     end
152                             end
153                     end
154             end
155     end
156
157     endmodule
```

Listing 16: uart.v

```
1   module seven_seg(
2       input [3:0] value,
3       output reg [6:0] seg = 0
4   );
5
6       // translate the bcd lookup value into the correct number, letter, or
7       // symbol if the display is not enabled just keep it blank default to
8       // a unused symbol to indicate an error
9
10      always @(value)
11          case (value)
12              4'h0: seg <= 7'b1000000;
13              4'h1: seg <= 7'b1111001;
14              4'h2: seg <= 7'b0100100;
15              4'h3: seg <= 7'b0110000;
16              4'h4: seg <= 7'b0011001;
17              4'h5: seg <= 7'b0010010;
18              4'h6: seg <= 7'b0000010;
19              4'h7: seg <= 7'b1111000;
20              4'h8: seg <= 7'b0000000;
21              4'h9: seg <= 7'b0010000;
22              4'hA: seg <= 7'b0001000;
23              4'hB: seg <= 7'b0000011;
24              4'hC: seg <= 7'b1000110;
25              4'hD: seg <= 7'b0100001;
26              4'hE: seg <= 7'b0000110;
27              4'hF: seg <= 7'b0001110;
28              default: seg <= 7'b1110110;
29          endcase
30
31  endmodule
```

Listing 17: seven_seg.v

```
1  `define ADDR_GPI_0        0
2  `define ADDR_GPI_1        1
3  `define ADDR_GPI_2        2
4  `define ADDR_GPI_3        3
5  `define ADDR_GPI_4        4
6  `define ADDR_GPI_5        5
7  `define ADDR_GPI_6        6
8  `define ADDR_GPI_7        7
9  `define ADDR_GPO_0        8
10 `define ADDR_GPO_1        9
11 `define ADDR_GPO_2        10
12 `define ADDR_GPO_3        11
13 `define ADDR_GPO_4        12
14 `define ADDR_GPO_5        13
15 `define ADDR_GPO_6        14
16 `define ADDR_GPO_7        15
17 `define ADDR_UART_RXD     16
18 `define ADDR_UART_TXD     17
19 `define ADDR_UART_CTL     18
20 `define ADDR_TMR_0        19
21 `define ADDR_TMR_1        20
22 `define ADDR_TMR_2        21
23 `define ADDR_TMR_3        22
24 `define ADDR_TMR_TRIG     23
25 `define ADDR_TMR_RST      24
26
27 `define RESET_VECTOR      32
28
29 `define OP_HALT       0
30 `define OP_LD         1
31 `define OP_ST         2
32 `define OP_STL        3
33 `define OP_LDL        4
34 `define OP_MOV        5
35 `define OP_ADD        6
36 `define OP_SUB        7
37 `define OP_AND        8
38 `define OP_OR         9
39 `define OP_XOR        10
40 `define OP_SFL        11
41 `define OP_SFR        12
42 `define OP_INC        13
43 `define OP_DEC        14
44 `define OP_EQL        15
45 `define OP_GTH        16
46 `define OP_LTH        17
47 `define OP_INV        18
48 `define OP_BRZ        19
49 `define OP_BRNZ       20
50 `define OP_JMP        21
51
52 `define RAM_ADDR_BITS    13
```

Listing 18: constants.v