In this first chapter we learn the very basics of a CPU's core logic. We include basic Function Unit logic such as logical operators and math functions. To keep our first chapter simple, branching is not yet implemented so our programs are strictly linear and cannot include loop constructs. Also, the program code is included directly in the CPU source file.

Author's Goals

I hope that you will follow along with me and create your own version of yfcpu. If you simply copy my code you will not truly learn what I am trying to teach in these articles. As I write this now, I have a theoretical understanding of CPU design. I hope to put into practice the theory I know and deepen my understanding of the subject. Thus I don't claim to be an expert in this field but my goal in the final chapters is to develop an efficient embeddable CPU with pipelining and data/instruction caches. I also hope to compile and run ucLinux with it inside an FPGA. I hope my learning process as I write these articles will help as I am not so far removed in experience as possibly the reader. I may return to earlier chapters and write footnotes or sidebars if new knowledge deems it appropriate.

If you are familiar with FPGAs, HDLs and basic CPU design, you can skip to page 3 for the YFCPU Verilog source code.

Why a custom CPU?

A custom CPU implementation allows you to customize the instruction set for your application specific operations. This would only be beneficial for embedded type applications. If you are creating a small mp3 player for market your custom CPU could contain instructions for fast audio decoding thus requiring a lower overall clock speed and thus less expensive chip production technology. Furthermore, the removal of unnecessary instructions will reduce the logic footprint and possibly decrease the overall logic delay and thus increase your allowable instruction cycle time. Finally, in our case we wish to learn how processors work internally and how better to accomplish this than making our own simple RISC based processor.

Hardware Description Languages (HDLs)

Hardware Description Languages is what we currently use to design the behavior of logic circuits. In the older days of Commodore 64's, early Intel's and other processors, processors were designed by hand, laying out the layers of an IC's substrate masks using regular drafting

techniques. Seymour Cray of Cray computes used #3 pencils! There were little or no computer aided design tools to help the chip developer. This method was tedious to say the least and few people had the patience and skills for such a task.

Thankfully, times have changed and designing custom processors is within reach of many hobby designers. Especially, if we already have computer programming experience, the switch is a simple paradigm change in thinking given the nature of software programs versus logic circuits. Generally, computers programs are executed serially and hardware descriptions are primarily parallel and near instantaneous, (disregarding sequential circuits for the moment).

HDLs give us a way to describe and implement a hardware design in a similar manner as developing a software program. We also have similar tools like compilers, debuggers and simulators.There are two predominate HDL language, Verilog and VHDL. We will use Verilog because it has a simpler syntax, some of which is similar to the C language.

FPGAs and ASICs

There are two mainstream technologies for implementing a real IC based on your hardware description, FPGA and ASIC. You get an Aplication Specific Integrated Circuit part by sending your hardware description and a lot of money to a IC manufacturer. ASIC is a more recent term for "making your own IC". An Field Programmable Gate Array is a much cheaper alternative for small quantities. An FPGA contains a bucket of unconnected logic gates and allows us to program the connectedness of these logic gates using the FPGAs built in routing gridwork. Most FPGAs also allows us to reprogram them many times or even "in the field" allowing our application to have the latest functionality and bug patches.

What are IP cores?

You may have already found opencores.org. This is a great site with many freely available CPU cores and other IP cores. An IP core is simply a specific implimentation of a logic unit, whether it be a CPU or peripheral or other logic device. The IP stands for intellectual property. From opencores.org you can download one or more of these IP cores and instantiate them inside an FPGA device creating your own application specific System-on-Chip (SoC). Some companies specialize in creating and selling IP cores for various tasks, and in one such case I found a company that created a hardware based Java processor. They did not sell real ICs, only the descriptive logic source code for implentation inside an FPGA or ASIC.

Computer Timing & Clock Speeds

Given that the same integrated process technology is used, so as to compare apples to apples, a CPU's maximum clock speed is determined by how much logic delay exists between the start of a CPU's single instruction cycle and when all the bits settle on the correct state or the "static state". This is obviously taken as worst case scenario as some simple instructions will have less delay than a complicated instruction, For example, a boolean logic operation versus a multiply or even worse a divide.

For illustrative purposes, imagine a complex domino effect with many branching dominos. As the cycle starts, each domino is flipped in sequence, though some spawn new branches which run in parallel, but in the end the longest domino path determines how long the domino cycle takes to reach the static state.

If you wish to increase the speed of your CPU you could remove the instructions with the longest execution delays. You could then increase the cycles per second until you are encroaching on the delay of the next longest instruction. However, instructions with long execution delays could be performing a useful function that would take a handful of simpler instructions to replace. The question is, will your software use that instruction enough to justify slowing down the clock speed ***all*** instructions? This is the argument between the pro CISC and the pro RISC groups.

RISC vs CISC

Reduced Instruction Set Computer or Complex Instruction Set Computer? Believe it or not, computer processor technology began as CISC type CPUs. Processors became more advanced by implementing more complex instructions that could do the same work as many simpler instructions. The trade-off of RISC versus CISC is more in favor of CISC when cpu's had a clock speed of only a few megahertz and compilers were very basic or assembly was used. Now that processing speeds are in the gigahertz RISC is winning the battle as a small savings in cycle time means significantly more instructions can be executed per second and intelligent compilers take more advantage of RISC type architecture. Modern x86 PCs though originally CISC are typically now RISC at the core with a CISC front-end and back-end conversion. This trend was started by AMD years ago to create competitive Intel compatible processors with less investment but has similarly now been adopted by Intel with Core 2 CPUs. Even RISC instruction sets are becoming more complex so the line between RISC and CISC is drawing closer.

See *Other Resources* below for more information on RISC and CISC architectures.

Designing our first basic CPU

Our first attempt at creating a cpu will be very simplistic. We will keep the entire cpu module in a single file, including the program assembler code.

Our basic CPU will execute a single instruction in 4 clock cycles. These four cycles are named Fetch, Decode, Execute and Store (aka. Write Back). We define tokens for these states starting at line 11. These four states define a Finite State Machine (FSM) implemented using a case statement starting at line 80 (same as switch/case in the C language.) The Fetch, Decode and Store states are very simple, but the Execute state, which comprises the cpu's ALU (Arithmetic and Logic Unit) is more complex and is implemented using another case statement (line 102). The decoded opcode is used by the ALU to determine the operation performed on the input data registers RA & RB. Each ALU operation, or instruction opcode, is declared starting at line 20.

The instruction memory and register file is declared at line 29. Global declared module parameters determine the size of these memories, namely rf_size, im_size and opcode_size. With rf_size equal to 4, this means we will use 4 bits to address the register file. Thus, our register file can and will contain 2^4 (=16) registers. The bit width of the instruction word is affected by the bit width of the register file fields. We have 4 bits for the opcode, three register file fields (RA, RB & RD) so our instruction word bit length is 16 bits (4+3x4).

We have only a single instruction word format so our instruction decoder is very simple, split the IR register into OPCODE, RA, RB & RD (line 93). An instruction word format defines which bits of the instruction word define fields such as registers, constants or other types. The format used by the decoder is defined typically by the opcode. A typical cpu may have 3 or 4 instruction word formats and in later chapters we will add more.

Here is a breakdown of the four cycles of our basic cpu.

FETCH: The next instruction pointed to by the PC register is loaded into the IR (Instruction Register) register.

DECODE: The PC (Program Counter) is incremented (could also have been done in the fetch cycle), and the IR register is decoded into the OPCODE register, and the input register addresses RA & RB, and the write back register RD. In later chapters, this decoder may use different instruction word formats.

EXECUTE: Switch/case on the defined instruction opcodes using the OPCODE register and perform the defined operation using the addressed data referenced by the input register fields RA & RB. In some cases, RA and RB may be taken as constants and not access the register file at all such as the opcode LRI (Load Register Immediate). The return value from the ALU is stored in the W register.

STORE: Write the W register (the return value from the ALU), into register file memory.

Remember, in our basic cpu, we have not included any branch instructions yet. So our program performs a few basic calculations and then quits. You can view the assembly code for the sample program implemented as a constant array starting at line 59. (Note. Verilog syntax defines the "4'd" prefix to constants to mean a decimal number of 4 bits wide, so don't let all those *4'd*nn prefixed numbers scare you, they are simply RA,RB,RD register addresses in decimal.)

We could probably improve the efficiency by doing more in the simple cycles and perhaps get down to 3 cycles per instruction, however this 4 cycle RISC cpu is a typical design and in later chapters we will expand on each cycle's duties and pipeline the cpu for single instruction per cycle execution.

Other Resources

Reduced Instruction Set Computers - Wiki

A CPU Datapath Simulator

Verilog source:

Chapter 1 - Basic CPU Logic (tgz)

Chapter 1 - Basic CPU Logic (windows zip)

yfcpu.v

```
  1:  module yfcpu(clk, rst, PC);
 2:
 3:  // our cpu core parameters
 4:  parameter im_size = 8;        // 2^n instruction word memory
 5:  parameter rf_size = 4;        // 2^n word register file
 6:
 7:  input clk;    // our system clock
 8:  output PC;    // our program counter
 9:  input rst;    // reset signal
10:
11:  // the cycle states of our cpu, i.e. the Control Unit states
12:  parameter s_fetch = 2'b00;    // fetch next instruction from memory
13:  parameter s_decode = 2'b01;    // decode instruction into opcode and operands
14:  parameter s_execute = 2'b10;    // execute the instruction inside the ALU
15:  parameter s_store = 2'b11;    // store the result back to memory
16:
17:  // the parts of our instruction word
18:  parameter opcode_size = 4;        // size of opcode in bits
19:
20:  // our mnemonic op codes
21:  parameter LRI  = 4'b0001;
22:  parameter ADD  = 4'b0100;
23:  parameter OR   = 4'b0110;
24:  parameter XOR  = 4'b0111;
25:  parameter HALT = 4'b0000;
26:
27:  // our memory core consisting of Instruction Memory, Register File and an ALU working
(W) register
28:  reg [ opcode_size + (rf_size*3) -1 : 0 ] IMEM[0: 2 ** im_size -1 ] ;    // instruction memory
```

```
29:  reg [ 7:0 ] REGFILE[0: 2 ** rf_size -1 ];    // data memory
30:  reg [ 7:0 ] W;    // working (intermediate) register
31:
32:  // our cpu core registers
33:  reg [ im_size-1 : 0 ] PC;           // program counter
34:  reg [ opcode_size + (rf_size*3) -1 : 0 ] IR;    // instruction register
35:
36:  /* Control Unit registers
37:       The control unit sequencer cycles through fetching the next instruction
38:       from memory, decoding the instruction into the opcode and operands and
39:       executing the opcode in the ALU.
40:  */
41:  reg [ 1:0 ] current_state;
42:  reg [ 1:0 ] next_state;
43:
44:  // our instruction registers
45:  // an opcode typically loads registers addressed from RA and RB, and stores
46:  // the result into destination register RD. RA:RB can also be used to form
47:  // an 8bit immediate (literal) value.
48:  reg [ opcode_size-1 : 0 ] OPCODE;
49:  reg [ rf_size-1 : 0 ] RA;   // left operand register address
50:  reg [ rf_size-1 : 0 ] RB;   // right operand register address
51:  reg [ rf_size-1 : 0 ] RD;   // destination register
52:
53:
54:  // the initial cpu state bootstrap
55:  initial begin
56:      PC = 0;
57:      current_state = s_fetch;
58:
59:      // PROGRAM MEMORY : initialize our instruction memory with a test program
60:      // IMEM[n] = { OPCODE, RA, RB, RD };
61:      IMEM[0] = { LRI , 4'd00, 4'd00, 4'd00 };   // clear R1, R2, R3
62:      IMEM[1] = { LRI , 4'd02, 4'd04, 4'd01 };   // load immediate into R1
63:      IMEM[2] = { LRI , 4'd01, 4'd11, 4'd02 };  // load immediate into R2
64:      IMEM[3] = { ADD, 4'd01, 4'd02, 4'd03 };   // add R1 + R2, into R3
65:      IMEM[4] = { XOR, 4'd02, 4'd03, 4'd04 };   // or R2 & R3 into R4
66:      IMEM[5] = { OR  , 4'd02, 4'd01, 4'd00 };   // or R2 & R1 into R0
67:      IMEM[6] = { HALT, 12'd0 };  // end program
68:      end
69:
70:  // at each clock cycle we sequence the Control Unit, or if rst is
71:  // asserted we keep the cpu in reset.
72:  always @ (clk, rst)
73:  begin
74:      if(rst) begin
```

```
75:         current_state = s_fetch;
76:         PC = 0;
77:         end
78:       else
79:         begin
80:         // sequence our Control Unit
81:         case( current_state )
82:             s_fetch: begin
83:                 // fetch instruction from instruction memory
84:                 IR = IMEM[ PC ];
85:                 next_state = s_decode;
86:                 end
87:
88:             s_decode: begin
89:                // PC can be incremented as current instruction is loaded into IR
90:                 PC = PC + 1;
91:                 next_state = s_execute;
92:
93:                 // decode the opcode and register operands
94:                 OPCODE = IR[ opcode_size + (rf_size*3) -1 : (rf_size*3) ];
95:                 RA = IR[ (rf_size*3) -1 : (rf_size*2) ];
96:                 RB = IR[ (rf_size*2) -1 : (rf_size  ) ];
97:                 RD = IR[ (rf_size  ) -1 : 0 ];
98:                 end
99:
100:             s_execute: begin
101:                // Execute ALU instruction, process the OPCODE
102:                 case (OPCODE)
103:                     LRI: begin
104:                        // load register RD with immediate from RA:RB operands
105:                        W = {RA, RB};
106:                        next_state = s_store;
107:                        end
108:
109:                    ADD: begin
110:                       // Add RA + RB
111:                        W = REGFILE[RA] + REGFILE[RB];
112:                        next_state = s_store;
113:                        end
114:
115:                    OR: begin
116:                       // OR RA + RB
117:                        W = REGFILE[RA] | REGFILE[RB];
118:                        next_state = s_store;
119:                        end
120:
```

```
121:                XOR: begin
122:                  // Exclusive OR RA ^ RB
123:                   W = REGFILE[RA] ^ REGFILE[RB];
124:                   next_state = s_store;
125:                   end
126:
127:                HALT: begin
128:                  // Halt execution, loop indefinately
129:                   next_state = s_execute;
130:                   end
131:
132:                // catch all
133:                default: begin end
134:              endcase
135:              end
136:
137:           s_store: begin
138:             // store the ALU working register into the destination register RD
139:              REGFILE[RD] = W;
140:              next_state = s_fetch;
141:              end
142:
143:         // invalid state!
144:           default: begin end
145:       endcase
146:
147:      // move the control unit to the next state
148:       current_state = next_state;
149:    end
150: end
151:
152: endmodule
```

Testbench Module

The yfcpu module is a hardware description. We havent instantiated it into real hardware yet, and in fact, we could instantiate it many times and have a multi-cpu IC. If we were targeting an FPGA we would want to instantiate physical IO pins and attach these pins to the inputs and outputs of our cpu module appropriately. However, for our initial tutorials we will simply be simulating the design in ModelSim or similar program so we will create a *testbench* source file instead. A testbench is a regular verilog source file, it instantiates our cpu module and will also provide our test stimuli to sequence our cpu. When designing any hardware module it is always a good idea to develop a robust testbench that fully tests all of it's the features for the desired response. Verilog includes some language constructs intended for testbench use only and these testbench features cannot be instantiated inside an FPGA. (A compiler error will result if the target is not set to a simulator.)

testbench.v

```
  1:
 2:  //
=================================================================================
====
 3:  // TESTBENCH FOR CPU CORE
 4:  //
=================================================================================
====
 5:
 6:  module tb ();
 7:
 8:  reg clk, rst;
 9:  wire [7:0] pc;
10:
11:  yfcpu mycpu (
12:   clk, rst, pc
13:  );
14:
15:  initial begin
16:   clk = 1;
17:   rst = 1;
18:   #1 rst = 0;
19:   #1300 rst = 0;
20:   $stop;
21:  end
22:
23:  always clk = #1 ~clk;
24:
25:  endmodule
```