Your First CPU - Chapter 4 - Pipelining

Electronics

Last Updated on Sunday, 16 May 2010 07:52

There are four cycles in our previous CPU design, Fetch, Decode, Execute and Store (aka 'Write back'). In our current cpu design these 4 cycles are performed one at a time, thus our CPU executes an instruction from memory for every 4 cycles of the CPU clock. With a few changes we can increase our performance substantially to almost achieve a single instruction per clock cycle.

There is a common analogy used when explaining CPU pipelining and also where the name pipelining is borrowed from. You can think of each of these cycles like steps in a manufacturing pipeline. Each step performs it's function and the result is passed to the next step. Like a toy that is assembled along a conveyor from station to station, each worker in turn adds a piece to the final assembly. In any moment, there is 1 toy being finished, and N-1 toys partially completed, where N is the number of assembly stations.

A similar semantic is used to reduce the 4 clock cycles per instruction (so N=4) down to an apparent single clock cycle per instruction, albiet with a few caveats we will discuss later. We turn the standard design on it's head, and each "station" of the pipeline performs a single operation each clock cycle on one of the N instructions in the pipeline. Each instruction still takes 4 clock cycles to fetch, decode, execute and write back, however each clock cycle will result in a finished instruction. Seen another way, at any particular instant 3 instructions will be in the process of executing and 1 will complete.

Pipeline Stalls

There are some problems that arise when implementing pipelining. These problems will prevent the CPU from reaching the theoretical maximum cpu speed of 1 instruction for each clock cycle. The problems arise most commonly during branching operations when a change in the PC (program counter) register occurs. In such cases there are partially finished instructions within the pipeline that assumed no branch will occur. In other words, instructions immediately after the branch instruction entered the pipeline and could not have advance knowledge of the outcome of the branch condition. Therefor, if and when a branch occurs, existing operations in the pipeline must be invalidated. There are numerous ways to handle these situations. For now, we will leave this up to the programmer or compiler to always put 3 NOP instructions immediately preceeding any branch instruction. These NOPS will ensure proper cpu behavior no matter the outcome of the branch condition. This essentially makes all branch instructions take 4 cycles, and all other instructions take a single cycle.

We also have issues with two consecutive instructions accessing the same memory if the first instruction intends to write a value back to that memory location. The second instruction may get an old value as it fetched it's data before the write operation finished. Again, as a programmer we can insert a NOP in these situations, or as a compiler writer we can catch them during code generation.

There are also hardware methods to handling pipleline stalls. For example, we can add a single bit to each pipleline step that indicates if the output of that step is valid. Then if the PC changes because of a branch instruction, we can set that bit to invalid consequently turning those pending operations into a NOP. We could also go further and duplicate part of the pipleline so that a branch traverses both outcomes until such time as the condition can be evaluated. The latter would keep with 1:1 instruction per clock timing at the cost of a more complex design requiring more logic.

Verilog Source

There are two areas in our design reguired to implement simple pipelining. (1) We must remove the outer state machine so instead of sequencing through the four instruction cycles each one is executed in parallel. (2) To link each cycle together like a pipelining assembly line we must also add some register storage in between to store the result from each cycle until the next cycle occurs.

```
module yfcpu(clk, rst, PC);

// our cpu core parameters
parameter im_size = 8;          // 2^n instruction word memory
parameter rf_size = 4;          // 2^n word register file

// a parameter for the bus width of our cpu
```

```
parameter bw = 8;

input clk;      // our system clock
output PC;      // our program counter
input rst;      // reset signal

// the cycle states of our cpu, i.e. the Control Unit states
parameter s_fetch   = 3'b000;   // fetch next instruction from memory
parameter s_decode  = 3'b001;   // decode instruction into opcode and operands
parameter s_execute = 3'b010;   // execute the instruction inside the ALU
parameter s_store   = 3'b011;   // store the result back to memory
parameter s_nostore = 3'b100; // dont store any result, skips a cycle

// the parts of our instruction word
parameter opcode_size = 4;              // size of opcode in bits

// Mnemonic Op Codes
parameter LRI   = 4'b0001;
parameter ADD   = 4'b0100;
parameter SUB   = 4'b0101;
parameter OR    = 4'b0110;
parameter XOR   = 4'b0111;

// our new branch mnemonics!
parameter BRA    = 4'b1000;   // branch to address in memory location RB
parameter BRANZ  = 4'b1001;   // branch to address in memory location RB, if RA is zero
parameter BRAL   = 4'b1010;   // branch to literal address RB
parameter BRALNZ = 4'b1011;   // branch to literal address RB, if RA is zero
parameter CALL   = 4'b1100;   // call subroutine; store current PC into RD and jump to address

parameter HALT  = 4'b1111;



// our memory core consisting of Instruction Memory, Register File and an ALU working (W) regi
reg [ opcode_size + (rf_size*3) -1 : 0 ] IMEM[0: (1 << im_size) -1 ] ;  // instruction memory
reg [ bw-1:0 ] REGFILE[0: (1 << rf_size) -1 ];  // data memory
reg [ bw-1:0 ] W;       // working (intermediate) register

// our cpu core registers
reg [ im_size-1 : 0 ] PC;                       // program counter
reg [ opcode_size + (rf_size*3) -1 : 0 ] IR;    // instruction register

/* Control Unit registers
     The control unit sequencer cycles through fetching the next instruction
     from memory, decoding the instruction into the opcode and operands and
     executing the opcode in the ALU.
*/
reg [ 2:0 ] current_state;
reg [ 2:0 ] next_state;

// our instruction registers
// an opcode typically loads registers addressed from RA and RB, and stores
// the result into destination register RD. RA:RB can also be used to form
// an 8bit immediate (literal) value.
reg [ opcode_size-1 : 0 ] OPCODE;
reg [ rf_size-1 : 0 ] RA;   // left operand register address
reg [ rf_size-1 : 0 ] RB;   // right operand register address
reg [ rf_size-1 : 0 ] RD;   // destination register
```

```
/* Pipeline flags */
reg F_STORE;    // execute cycle sets if W should be written to RD in writeback cycle
reg F_HALTED;   // if set, we halt execution until a reset instruction (in a sim, we finish)

// the initial cpu state bootstrap
initial begin
        PC = 0;
        current_state = s_fetch;

        // LOADROM: here we load the rom file generated by our assembler!!!
        $readmemh("test.rom",IMEM,0,14);

        wait(OPCODE == HALT) $writememh("regfile",REGFILE);
        end




// at each clock cycle we sequence the Control Unit, or if rst is
// asserted we keep the cpu in reset.
always @ (clk, rst)
begin
        if(rst) begin
                current_state = s_fetch;
                PC = 0;
                F_HALTED = 0;
                end
        else if(!F_HALTED)
           begin
           // perform a fetch, decode, execute and writeback cycle concurrently
                // FETCH :  fetch instruction from instruction memory
                IR = IMEM[ PC ];
                PC = PC + 1;    // increment program counter (instruction mem pointer)

                // DECODE : decode the opcode and register operands
                OPCODE = IR[ opcode_size + (rf_size*3) -1 : (rf_size*3) ];
                RA = IR[ (rf_size*3) -1 : (rf_size*2) ];
                RB = IR[ (rf_size*2) -1 : (rf_size  ) ];
                RD = IR[ (rf_size  ) -1 : 0 ];

                // EXECUTE : Execute ALU instruction, process the OPCODE
                case (OPCODE)
                        LRI: begin
                           // load register RD with immediate from RA:RB operands
                           W = {RA, RB};
                           F_STORE = 1;
                           end

                        ADD: begin
                           // Add RA + RB
                           W = REGFILE[RA] + REGFILE[RB];
                           F_STORE = 1;
                           end

                        SUB: begin
                           // Sub RA - RB
                           W = REGFILE[RA] - REGFILE[RB];
                           F_STORE = 1;
                          end

                        OR: begin
```

```
                    // OR RA + RB
                    W = REGFILE[RA] | REGFILE[RB];
                    F_STORE = 1;
                    end

                XOR: begin
                    // Exclusive OR RA ^ RB
                    W = REGFILE[RA] ^ REGFILE[RB];
                    F_STORE = 1;
                    end

                HALT: begin
                    // Halt execution, loop indefinately
                    F_HALTED = 1;
                    $finish;     // quit the sim
                    end

                BRA: begin
                    // branch to REGFILE[RB]
                    PC = REGFILE[RB];
                    F_STORE = 0;
                    end

                BRAL: begin
                    // branch to RB
                    PC = RB;
                    F_STORE = 0;
                    end

                BRANZ: begin
                    // branch to REGFILE[RB] if REGFILE[RA] is zero
                    if(REGFILE[RA] !=8'd0)
                        PC = REGFILE[RB];
                    F_STORE = 0;
                    end

                BRALNZ: begin
                    // branch to RB if REGFILE[RA] is zero
                    if(REGFILE[RA] !=8'd0)
                        PC = RB;
                    F_STORE = 0;
                    end

                CALL: begin
                    // call a subroutine; store PC into RD and jump to location in REGF
                    W = PC;
                    PC = RB;
                    F_STORE = 1;
                    end

                    // catch all : NOP!
                default: begin
                    F_STORE = 0;
                end
            endcase

            // STORE : store the ALU working register into the destination register RD
            if(F_STORE)
                    REGFILE[RD] = W;

    // move the control unit to the next state
```

```
        end
end

endmodule
```

Comments (16)

Can't find test.asm file

16. Thursday, 24 April 2014 07:16
(Sandeep)

Can't find the test.asm file, can someone please guide regarding it.

More and Missing files.

15. Monday, 05 August 2013 03:27
(EllisGL)

I definitely want to see more tutorials on this. On another note, all the files seem to be missing.

Make pipelined

14. Saturday, 10 December 2011 23:57
(Cameron Martens)

In order to make the above code pipelined, you need to reverse the order of your stages.
Currently they are:
- Fetch
- Decode
- Execute
- Writeback

You need to write them in your code as:
- Writeback
- Execute
- Decode
- Fetch

This way, your internal registers will be changed after their use, instead of before. Then it will be pipelined.

Pipeline Synthesis!

13. Sunday, 13 March 2011 18:32
(Veera)

What should i do if i have to synthesis this above piece of code on the FPGA?

No pipeline

12. Wednesday, 23 February 2011 21:23
(Fili)

Sorry, but I agree with Brent: no pipeline here. You just have a big block that executes a whole instruction, at a very slow clock.
Here is what should happen in a pipelined processor: (@1 means time, I2 means second instruction, 0 means idle)
@1: Fetch: I1, Decode: 0, Exec: 0, Write: 0
@2: Fetch: I2, Decode: I1, Exec: 0, Write: 0
@3: Fetch: I3, Decode: I2, Exec: I1, Write: 0
@4: Fetch: I4, Decode: I3, Exec: I2, Write: I1
@5: Fetch: I5, Decode: I4, Exec: I3, Write: I2
@6: Fetch: 0, Decode: I5, Exec: I4, Write: I3
@7: Fetch: 0, Decode: 0, Exec: I5, Write: I4
@8: Fetch: 0, Decode: 0, Exec: 0, Write: I5

On the other hand, your processor does this:
@1: Fetch: I1, Decode: I1, Exec: I1, Write: I1
@2: Fetch: I2, Decode: I2, Exec: I2, Write: I2
@3: Fetch: I3, Decode: I3, Exec: I3, Write: I3
@4: Fetch: I4, Decode: I4, Exec: I4, Write: I4
@5: Fetch: I5, Decode: I5, Exec: I5, Write: I5

To make it pipelined, all stages should be running at the same time. Put them in separate always blocks and use special transfer registers to move data from one stage to another. This way, if your stages take the times T1, T2, T3, T4, instead of (T1+T2+T3+T4) you will get Max(T1,T2,T3,T4) execution cycle (which should be much faster).

Pipelined?

11. Monday, 21 February 2011 22:13
(Brent)

I am sorry. I do not see how this is pipelined. Where are your pipeline registers? Everytime you have a clock transition, it is running all the code in the always block.

Pipelined

10. Saturday, 18 December 2010 12:16
Guru

Richards: you're right that you should see a ripple effect. I remember running into this when I first converted the code to pipelined, so perhaps I posted the wrong code?!? I will have to install modelsim again and check it out. I've been using Icarus verilog in order to stay open source. Thanks.

YFASM on 64-bit gcc

9. Friday, 03 December 2010 21:25
(Fraser Doswell)

The size of a long unsigned on a 64-bit pc is 8 bytes instead of 4.

Two places required a different value of the empty memory value from 0xff to 0xffUL or 0xffffffffUL

They are the memset calls in alloc_imem and the comparison in the gen procedure.

This worked in the gen procedure:

if(lpimem[sys.base] != 0xffffffffffffffffUL) with the memset constants above.

At this point I'm wondering why the comparison didn't work with the 4 byte value used in the memset calls. But it works now!

oops!

8. Friday, 03 December 2010 09:44
(RichardS)

Re: " It's not the case that IR is from instruction N, OPCODE from instruction N-1, RA,B and D from instruction N-2 etc. "

I realise there's a mistake in there: OPCODE, RA,B and D get there values from the same execution cycle.

Pipelined??

7. Friday, 03 December 2010 09:31
(RichardS)

The key word in my comment was "complete". I ran your design through ModelSim using your supplied test.asm. The first 4 instruction are:

LRI y, 11 # 11 -> register[5]
LRI x, 3 # 3 -> register[4]
LDR dec, 1 # 1 -> register[3]
LRI lc, 10 # 10 -> register[2]

After the first clock change I see 11 in register[3], after the second I see 3 in register[4] etc.

I was expecting to see a 'ripple' effect through the execution stages such that 11 got stored to register[5] on the 4th clock change.

Looking at it another way, ModelSim shows that at any clock change the values in IR, OPCODE, RA, RB, RD, and W are those from the one instruction being executed at that time. It's not the case that IR is from instruction N, OPCODE from instruction N-1, RA,B and D from instruction N-2 etc.

As I said I'm just learning this stuff and I hope you intend to continue posting on the subject.

Pipelined code changes

6. Thursday, 02 December 2010 17:45
Guru

If you look at the code difference between my pipelined version and the previous version you will see that the Fetch, Decode, Execute and Write-Back used to be a sequential state machine. (Done by a counter and a case statement.) In the pipelined version the counter and case statement was removed leaving each step to execute in parallel. Easy! I had to add a few registers then so the steps wouldnt overwrite each others data. You will notice that each step sets registers that are consumed by the next step, specifically:
FETCH: IMEM -> IR
DECODE: IR -> OPCODE,RA,RB,RD
EXECUTE: OPCODE+RA,RB,RD -> W
STORE: W -> REGFILE
Much of the details is inferred by the verilog synthesizer. For example, that the behavior specification requires that the REGFILE be dual-port RAM access with simultaneous write. This is not specified anywhere in the verilog code but if you looked at Xilinx log output for example you should see it. Also, REGFILE is always accessed through RA and RB registers only, if I made an instruction that used RD to access REGFILE, then the RAM would have to be tri-port which may not be synthesizable in most programmable logic. The RD register is thus always and only used for write back address.

Yes, it is piplelined!

5. Thursday, 02 December 2010 17:30
Guru

You are exactly right that one instruction executes per cycle. Did you expect more than 1 instruction to execute per cycle? That would not be pipelining but something along the lines of hyperthreading, multiple ALU blocks, SIMD or MIMD.

A pipelined cpu with only one of each of the cpu sub-blocks such as instruction decoder, data/addr buses, Arithmic/Logic Units (ALU) and RegFile can only ever execute 1 instruction per second (1 MIP/MHz). The reason for this is obvious, a single cpu sub-block can only logically process 1 instruction per clock cycle. The instruction decoder can only decode 1 instruction, The ALU can only compute 1 result, the write-back can only write back a single result, the buses can only handle a single data word, etc. If we duplicate cpu sub-blocks, then yes, we can get multiple instructions per cycle, but that's not pipelining. For example, Modern CPUs, and now many microcontrollers have multiple Multiply-Accumlate (MAC) modules, this allows many MAC operations to happen at the same time. This is called Single-Instruction-Multiple-Data or SIMD. Also, 1 instruction per cycle is a theoretical max, in reality it is slightly less than 1 due to pipeline stalls due to branches, etc.

For any instruction to be executed, it must go through a minimum of 4 cycles: Fetch, decode, Execute, Write-Back. A non-pipelined cpu will thus take 4 cycles per instruction. However, during each clock cycle only 1 of the 4 sub blocks are in use. Thus the design is not very efficient but it does take less logic since the sub blocks can share registers. In a pipelined cpu we ensure each sub block performs work every cycle, but we must add registers to store intermediate values which costs us some logic. But we still can't achieve more than a 1:1 instructions per clock cycle.

I think where you are confused is you probably thought a non-piplelined cpu would execute a single instruction per cycle. Take a look at the Microchip PIC datasheets, they clearly state that the number of instructions you excute is 1/4 the MHz of the clock. Atmel is different, they get 1/1 because thier mcu's are piplelined!

Pipelined??

4. Thursday, 02 December 2010 14:06
(RichardS)

I'm only a beginner at this stuff but surely this version is not pipelined. All it's doing is executing one complete instruction per clock change. I can't see any evidence of pipeline stages.

ISE Webpack - Simulation Worked!

3. Monday, 22 November 2010 21:54
(Fraser Doswell)

The yfcpu pipelined version worked in Xilinx ISE Webpack 12.2 usin ISim.

ISE likes ansi-style module parameters, and won't read the memory file for synthesis.

The program will have to be loaded into fpga block ram when the fpga is configured or into external ram once the fpga is programmed.

Pipelined Version

2. Monday, 22 November 2010 16:14

(Fraser Doswell)

Found the problem - it was a typo
Typed the code for better comprehension, instead of just copying and pasting.

The code from this page worked properly.

Pipelined Version Not Looping

1. Saturday, 20 November 2010 20:45
(Fraser Doswell)

Due to my error or something else, the pipelined yfcpu does not loop in icarus verilog.
The branching version without pipeline did loop as expected.

The pipelined cpu stops after the first loop.
More troubleshooting is required, but did anybody get your pipelined cpu to simulate the whole loop?

Fraser

the gtkwave display

Add your comment