A Tiny Computer

Chuck Thacker, MSR
3 September, 2007

Alan Kay recently posed the following problem:

“I'd like to show JHS and HS kids "the simplest non-tricky architecture" in which simple gates and flipflops manifest a programmable computer”.

Alan posed a couple of other desiderata, primarily that the computer needs to demonstrate fundamental principles, but should be capable of running real programs produced by a compiler. This introduces some tension into the design, since simplicity and performance sometimes are in conflict.

This sounded like an interesting challenge, and I have a proposed design. The machine is a Harvard architecture (separate data and instruction memory) RISC. It executes each instruction in two phases correspond to instruction access and register access and ALU transit. Registers are written at the end of the instruction. Figure 1 is a block diagram of the machine.

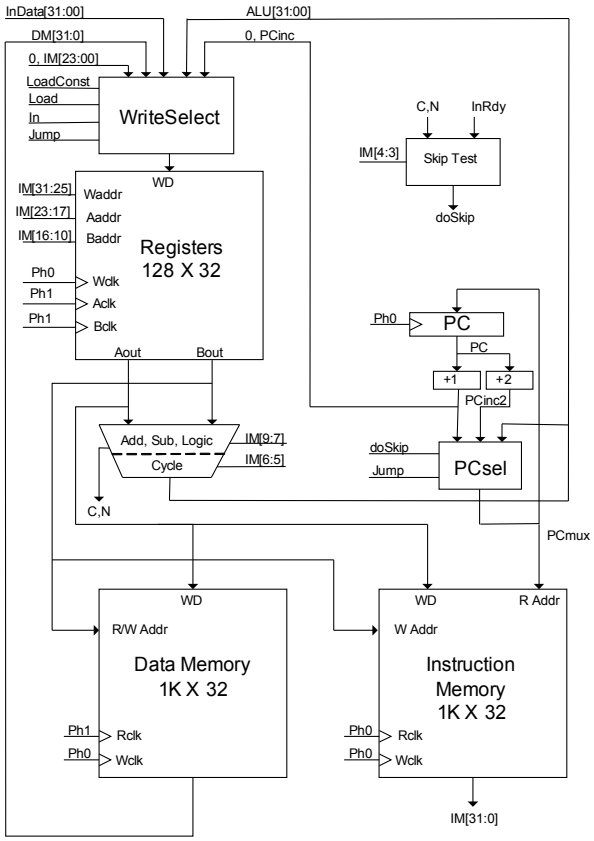

Figure 1: The Tiniest Computer?

Discussion and Implementation

Although it is impractical today to build a working computer with a "handful of gates and flipflops", it seemed quite reasonable to implement it with an FPGA (field programmable gate array). Modern FPGAs have enormous amounts of logic, as well as a number of specialized "hard macros" such as RAMs. Xilinx sells evaluation boards for about $150 that includes an FPGA and some auxiliary components for connecting the chip to real-world devices and the PC that runs the design tools (which are free to experimenters). This was the approach I took.

Although the machine was designed primarily for teaching, it may have other uses. It is small, fast, and has 32-bit instructions. This may make it competitive with more complex FPGA CPUs. The later section on "Extensions" describes some possibilities for making it a "real" computer (albeit one without Multiply/Divide, Floating Point arithmetic, or virtual memory).

I chose a Harvard architecture because in this arrangement, it is possible to access the data and instruction memories simultaneously. It is still possible to write self-modifying code (although this is usually considered a bad idea), since stores into both memories are supported.

Because it is implemented in the latest generation semiconductor technology (65 nm), the design uses a Xilinx Virtex-5 device. This part has an interesting feature that contributes to the small size of the overall design – a dual-ported static RAM with 1024 words of 36 bits. This RAM is used for the data and instruction memories, and two of them are used to provide the triple-ported register file.

The machine has 32-bit data paths. Most "tiny" computers are 8 or 16 bits wide, but they were designed originally in an era in which silicon was very expensive and package pins were scarce. Today, neither consideration applies. We will implement a variant of the machine with 36-bit data paths.

The design on the instruction set for the machine was determined primarily by the instruction and data path widths. It is a RISC design, since that seemed to be the simplest arrangement from a conceptual standpoint, and it is important to be able to explain the operation clearly.

Although the memories are wide, they are relatively small, containing only 1K locations. The section on extensions discusses some ways to get around this limit. For pedagogical purposes, and for the immediate uses we envision, a small memory seems adequate.

The memory is word-addressed, and all transfers involve full words. Byte addressing is a complexity that was introduced into computers for a number of reasons that are less

relevant today than they were thirty years ago. There is very limited support for byte-oriented operations.

One thing that is quite different even from modern machines is that the number of registers directly accessible to the program is 128. This number was chosen because the three register addresses fit comfortably into a 32-bit instruction. The value of this many registers may seem questionable, but given the implementation technology, they are extremely cheap. This was not the case when most computers in use today were designed. In addition, there is no significant performance advantage in using fewer registers. The instruction and data memories can be accessed in a single cycle, and so can the register file. A register file of 128 words uses only 1/8 of the block RAM that implements it. The Extensions section discusses ways in which the remaining registers might be used. It might be argued that a compiler cannot effectively use this many registers. This was certainly true for a compiler designed in an era in which registers were expensive and much faster than the main store. This is not the case here, and it will be interesting to see whether a compiler that uses whole-program analysis can actually use this many registers. If they turn out to be unnecessary, it is easy to reduce the number of registers.

The only discrete register in the design is the program counter (PC). PC is currently only 10 bits wide, but it could easily expand to any length up to 32 (or 36) bits. The memories used for RF, IM, and DM all have registers inside them, so we don't need to provide them. We do need the PC, since there is no external access to the IM read address. PC is a copy of this register.

The instruction set (Figure 2) is very simple and regular. All instructions have the same format. Most operations use three register addresses, and most logical and arithmetic instructions are of the form Rw <- function(Ra, Rb). This seemed easier to explain than a machine that used only one or two register addresses per instruction. It also improves code density and reduces algorithm complexity. If the LC (load constant) bit is true, the remaining low order bits of the instruction are treated as a 24-bit constant (zero extended) and written to Rw. Any Skip or Jump is suppressed. The In, LoadDM, and Jump instructions load Rw with data from an input device, data from DM[Rb], or PC + 1. All other instructions load Rw with F(Ra, Rb). Any instruction except Jump conditionally skips the next instruction if the condition implied by the SK field is true. The StoreIM and StoreDM instructions do DM/IM[Rb] <- Ra. These instructions also load Rw with the ALU result, and can also conditionally skip. The Output instruction simply generates a strobe. The intent is that Ra is output data, Rb is an output device selector, but other arrangements are possible.

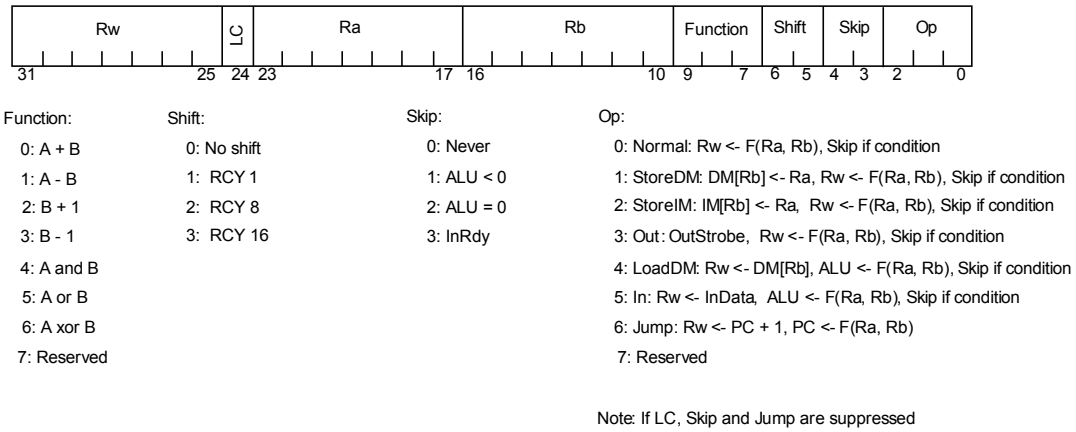

Figure 2: Instruction Format

There are relatively few ALU functions. The shifter is placed after the adder/subtractor/logic unit. It too has limited capabilities. If we need more elaborate arithmetic, we can add one or more of Xilinx' DSP48E cores to the design. These devices are high speed MACs designed for signal processing, but they can do a number of other operations. If we want to avoid DSPs, we need ways to do multiple precision adds and subtracts, multiply, and divide, preferably at one operation per result bit. There are well-known ways to do this while not increasing the complexity too much. But they need more Function and Shift bits to specify them. These could be had by reducing the number of registers to 64, or by increasing all data path widths to 36 bits. The modification needed to support these operations is trivial.

The machine executes instructions in two phases (Ph0 and Ph1). This is unlike essentially all modern computers, which use pipelining to improve performance. The phases are different lengths, since during phase 0, we only need to access the IM, while during phase 1, we must read from the register file, do the ALU operation, and test the result to determine whether the instruction skips. This takes much longer then simply accessing a register. This also makes it much easier to explain how the machine functions. Even with this simplification, the performance is adequate, executing about 60 million instructions per second.

This approach was first employed (I believe) in the original Data General Nova, a simple machine that still has a lot to teach us about computer architecture, since it was arguably the first commercial RISC machine. The major differences are:

(1) There are more registers (128 vs. 4)
(2) There are three register select fields instead or two.
(3) The Function field has different meanings.
(4) The Nova's Carry field has been eliminated.
(5) The Skip field is different.
(6) There is no "No load" bit.

The Jump instruction saves the (incremented) PC in Rw. This is the only support for subroutines. There is no call stack. Programs that need a stack must construct it themselves.

There is an operation (LC) to load a 24-bit constant (with leading zeros) into Rw. Fabricating constants is usually difficult on a machine with only a few short constants. During an instruction that loads a constant, skips and jumps are suppressed.

There is very little support for input/output. All I/O is programmed, transferring a single 32-bit word between the machine and an external device. There are no interrupts (but see the section on extensions for a possible way of dealing with this). Because the instruction time is completely deterministic, it is easy to write timing-critical programs.

The register addressed by Rw is always written at the end of the instruction. If the value produced is unwanted, Rw should point to a "trashcan" register.

The instruction sequencing of the machine is shown in Figure 3.

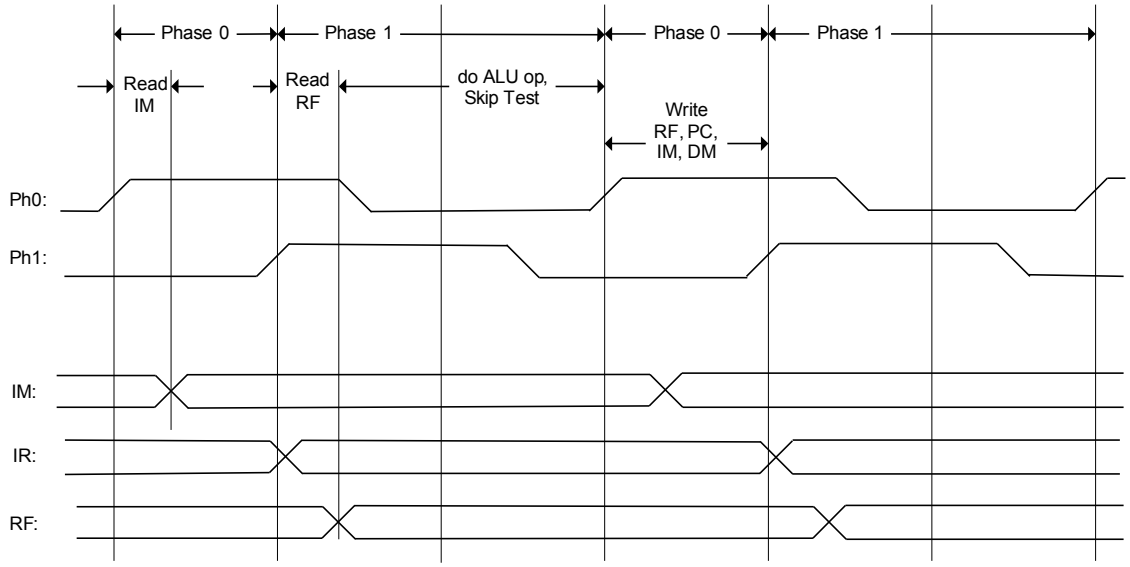

Figure 3: Instruction Timing

Each instruction requires two phases

During the first phase, IM is first read to retrieve the instruction. At the beginning of phase 1, the RF address register is loaded with the Ra and Rb addresses specified in the instruction. RF is read to retrieve the operands. When RF's outputs appear, they are passed through the ALU and finally, the register file and PC are written to the register file at the end of the instruction.

The time to read RF (T_{RCK_DO}) plus the time to do an ALU operation and test the result is the reason for the asymmetry, since this is a long combinational path. Phase 0 is approximately of length $T_{RCK\_DO} + T_{RCCK\_ADDR}$ of the Block RAM, plus wiring delay.

The machine is started at Reset by preloading the instruction memory (and if necessary, the data memory) with a bootstrap loader. This is done as part of configuring the FPGA, and the loader is included in the FPGA's bitstream. The PC is reset to 0, and the loader begins executing. It can load the remainder of the IM and the DM from an I/O device or external RAM/ROM.

Size and Speed

In Virtex-5 technology, the machine occupies about 200 LUTs (lookup tables) and four block RAMs, although a more complex ALU would increase this It runs at 66 MHz, although this could doubtlessly be improved somewhat at the expense of more time spent routing the design. The Verilog describing the entire design is two pages long (Appendix A).

Extensions

The limited size of DM and IM is the main thing that makes this computer noncompetitive. This could be mitigated by using the memories as caches rather than RAM. The 1K BRAM holds 128 eight-word blocks, which is the transfer size of the DRAM controller we are designing for the BEE3. We would need to provide I and D tag stores, but this wouldn't be very expensive.

For our immediate application, we plan to use the processor to initialize and test two DDR2 memory controllers that connect to two off-FPGA 4 GB DIMMs each (for a total of 16 GB). Each DDR DIMM contains two 2 GB ranks of DRAMs, so there are four ranks per controller. Since these controllers transfer 36 bytes (8 words of 36 bits) in one access, we can use the write port of DM (which is otherwise unused) as the source and destination of DRAM data. To do an operation, the system will load three output registers: A command/rank register containing 3 bits of command, the controller select (1 bit) and the DIMM rank (2 bits), a DRAM address register (31 bits), and the DM address that will supply or receive the data (10 bits). Loading the command starts the transfer, and when it is complete, the data will have been taken from or written to eight successive locations in DM. Completion is tested by testing a "done" flag that is cleared when the command register is loaded, and set when the transfer is complete. As an added feature, we can use the reserved ALU function to generate the next sequence in a prime-polynomial linear feedback shift register. This will be used to generate pseudo-random patterns to test the DIMMs. For this application, the data paths will be increased in width to 36 bits.

The second extension addresses the lack of interrupts. Since the BRAM holding the registers can hold eight full register contexts, it should be straightforward to provide a mechanism similar to that of the Alto. A separate three-bit Context register would hold the current context. Saving and restoring the current context's PC on a task switch is a bit problematic, but it is probably fairly straightforward.

Appendix A: Tiny Computer Verilog description

```
`timescale 1ns / 1ps
module TinyComp(
 input Ph0In, Ph1In, //clock phases
 input Reset,
 input [31:00] InData, // I/O input
 input InRdy,
 output InStrobe, //We are executing an Input instruction
 output OutStrobe //We are executing an Output instruction
);

wire doSkip;
wire [31:00] WD; //write data to the register file
wire [23:00] WDmid;  //the WD mux intermediate outputs
wire [31:00] RFAout; //register file port A read data
wire [31:00] RFBout; //register file port B read data
reg  [9:0]   PC; //10-bit program counter
wire [9:0]   PCinc, PCinc2, PCmux;
wire [31:00] ALU; // ALUoutput
wire [31:00] AddSubUnit;
wire [31:00] ALUresult;
wire [31:00] DM; //the Data memory (1K x 32) outputs
wire [31:00] IM; //the Instruction memory (1K x 32) outputs
wire Ph0, Ph1; //the (buffered) clocks
wire WriteIM, WriteDM, Jump, LoadDM, LoadALU; //Opcode decodes
//----------End of declarations------------

//this is the only register, other than the registers in the block RAMs.
// Everything else is combinational.
 always @(posedge Ph0)
  if(Reset) PC <= 0;
  else PC <= PCmux;

// the Phases.  They are asymmetric -- see .ucf file
BUFG ph0Buf(.I(Ph0In), .O(Ph0)); //this is Xilinx - specific
BUFG ph1Buf(.I(Ph1In), .O(Ph1));

//the Skip Tester. 1 LUT
assign doSkip = (~IM[24] & ~IM[4] &  IM[3] & ALU[31]) |
                (~IM[24] &  IM[4] & ~IM[3] & (ALU == 0)) |
                (~IM[24] &  IM[4] &  IM[3] & InRdy);

//Opcode decode.  7 LUTs
 assign WriteIM   = ~IM[24] & ~IM[2] & ~IM[1] &  IM[0]; //Op 1
 assign WriteDM   = ~IM[24] & ~IM[2] &  IM[1] & ~IM[0]; //Op 2
 assign OutStrobe = ~IM[24] & ~IM[2] &  IM[1] &  IM[0]; //Op 3
 assign LoadDM    = ~IM[24] &  IM[2] & ~IM[1] & ~IM[0]; //Op 4
 assign InStrobe  = ~IM[24] &  IM[2] & ~IM[1] &  IM[0]; //Op 5
 assign Jump      = ~IM[24] &  IM[2] &  IM[1] & ~IM[0]; //op 6
 assign LoadALU  =  ~IM[24] & ~IM[2] ;                  //Ops 0..3

// instantiate the WD multiplexer. 24*2 + 8 = 56 LUTs
 genvar i;
 generate
  for(i = 0; i < 32; i = i+1)
  begin: wsblock
   if(i < 10 )begin
    assign WDmid[i] = (LoadALU & ALU[i]) | (InStrobe & InData[i]) | (LoadDM & DM[i]);
//6-in
    assign WD[i] = (Jump & PCinc[i]) | (IM[24] & IM[i]) | WDmid[i]; //5-in
  end else if(i < 24) begin
    assign WDmid[i] = (LoadALU & ALU[i]) | (InStrobe & InData[i]) | (LoadDM & DM[i]);
//6-in
    assign WD[i] = (IM[24] & IM[i]) | WDmid[i]; //3-in
  end else
    assign WD[i] = (LoadALU & ALU[i]) | (InStrobe & InData[i]) | (LoadDM & DM[i]); //6-in
  end //wsblock
endgenerate

//the PC-derived signals
```

```
 assign PCinc =  PC + 1;
 assign PCinc2 = PC + 2;
 assign PCmux = Jump ? ALU[9:0] : doSkip ? PCinc2 : PCinc;

//instantiate the IM. Read during Ph0, written (if needed) at the beginning of the next
Ph0
 ramx im(
   .clkb(Ph0 ), .addrb(PCmux[9:0]),                .doutb(IM),     //the read port
   .clka(Ph0), .addra(RFBout[9:0]), .wea(WriteIM), .dina(RFAout)); //the write port

//instantiate the DM. Read during Ph1, written (if needed) at the beginning of the next
Ph0
 ramx dm(
   .clkb(Ph1), .addrb(RFBout[9:0]),                .doutb(DM),  //the read port
   .clka(Ph0), .addra(RFBout[9:0]), .wea(WriteDM), .dina(RFAout)); //the write port

//instantiate the register file.  This has three independent addresses, so two BRAMs are
needed.
// read after the read and write addresses are stable (rise of Ph1) written at the end of
the
// instruction (rise of Ph0).
 ramx rfA(.addra({3'b0, IM[31:25]}), .clka(Ph0), .wea(1'b0), .dina(WD), //write port
   .clkb(Ph1), .addrb({3'b0, IM[23:17]}), .doutb(RFAout));              //read port
 ramx rfB(.addra({3'b0, IM[31:25]}), .clka(Ph0), .wea(1'b1), .dina(WD), //write port
   .clkb(Ph1), .addrb({3'b0, IM[16:10]}), .doutb(RFBout));              //read port

//instantiate the ALU: An adder/subtractor followed by a shifter

//32 LUTs. IM[8] => mask A, IM[7] => complement B, insert Cin
 assign AddSubUnit = ((IM[8]? 32'b0 : RFAout) + (IM[7] ? ~RFBout : RFBout)) + IM[7];
//generate the ALU and shifter one bit at a time
 genvar j;
 generate
  for(j = 0; j < 32; j = j+1)
   begin: shblock
    assign ALUresult[j] =  //32 LUTs
     (~IM[9] & AddSubUnit[j]) |  //0-3: A+B, A-B, B+1, B-1
     ( IM[9] & ~IM[8] & ~IM[7] & (RFAout[j] & RFBout[j])) | //4: and
     ( IM[9] & ~IM[8] &  IM[7] & (RFAout[j] | RFBout[j])) | //5: or
     ( IM[9] &  IM[8] &  IM[7] & (RFAout[j] ^ RFBout[j])) ; //6: xor

    assign ALU[j] = //32 LUTs
     (~IM[6] & ~IM[5] & ALUresult[j]) | //0: no cycle
     (~IM[6] &  IM[5] & ALUresult[(j + 1)  % 32]) | //1: rcy 1
     ( IM[6] & ~IM[5] & ALUresult[(j + 8)  % 32]) | //2: rcy 8
     ( IM[6] &  IM[5] & ALUresult[(j + 16) % 32]) ; //rcy 16
   end //shblock
 endgenerate
endmodule
```