Chuck Thacker

A Tiny Computer

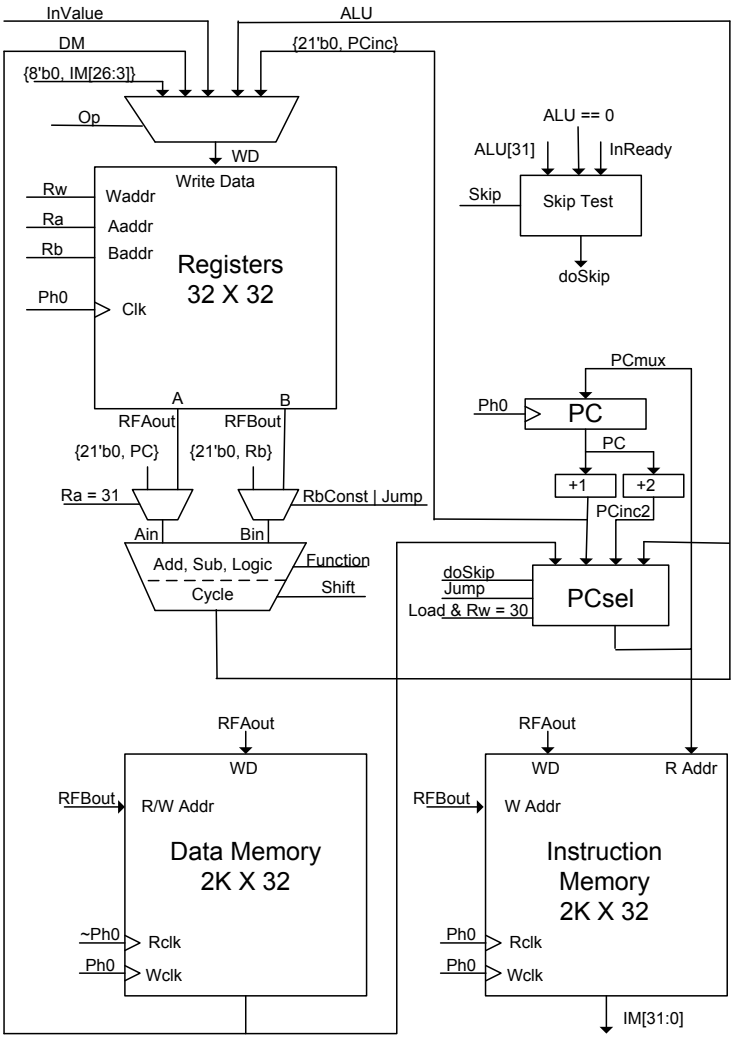
In late 2007, Alan Kay said to me: "I'd like to show junior and senior high school kids the simplest non-tricky architecture in which simple gates and flip-flops manifest a programmable computer."

Alan posed a couple of other desiderata, primarily that the computer needed 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 are sometimes in conflict.

This sounded like an interesting challenge, so I designed the machine shown below.

Implementation

Although it is impractical today to build a working computer with a "handful of gates and flip-flops," 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. The basic logic is provided by lookup tables (LUTs) that can be configured to produce any Boolean function of their inputs. Each LUT is accompanied by a flip-flop that may be used to construct registers. All wiring between the LUTs and registers, as well as the functions done by the LUTs, is configurable by a



"bit stream" file, which is loaded into the chip at initialization. The Spartan-3 part used for the TC had 4-input LUTs. More modern FPGAs have six-input LUTs.

Xilinx Corporation sells evaluation boards for about $150 that include an FPGA, and some auxiliary components for connecting the chip to real-world

devices and to a PC that runs the design tools, which are free to experimenters. This was the approach I selected. The evaluation board was the Spartan-3E Starter Kit. There are similar evaluation boards available today that contain much larger FPGAs.

These days, hardware is designed by doing programming. It's just like writing a program to calculate sums, except that the output of the program is a specification of what the system being designed should do, rather than the immediate result. When you write "`x <= A+B`" you are not asking for the value of `x`, you're asking for an adder, which can give you the value of `x` for any `A` and any `B`. The Verilog synthesizer politely produces an adder.

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 section below on extensions describes some possibilities for making it a "real" computer, albeit a fairly limited one. The evaluation board provides a number of interesting components that could also be used to extend the design.

I chose a "Harvard" architecture with separate data and instruction memories, because 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 writes into both memories are supported.

At the time the design was done, the Spartan-3E was the most cost-effective FPGA made by Xilinx, but it was implemented in a 90nm silicon process which was already obsolete by one generation. Today's FPGAs are implemented in a 45nm process, so they are both faster and less expensive. Xilinx FPGAs have an interesting feature that contributes to the small size of the overall design—a dual-ported static RAM with 1024 words of 18 bits. This RAM is used for the data and instruction memories (four for each memory). The register file uses memories built from the FPGA's lookup tables.

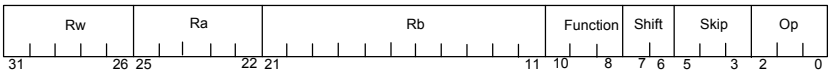
The machine has 32-bit data paths. Most "tiny" computers are 8 or 16 bits wide, but they were designed originally in an era in which silicon was very expensive and package pins were scarce. Today, neither consideration applies.

The design of 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 2048 locations. The section on extensions discusses some ways to get around this limit. For pedagogical purposes, a small memory seemed 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.

The primary discrete register in the design is the program counter (PC). PC is currently only 11 bits wide, but it could easily expand. The memories used for IM and DM register their addresses and input data, so we don't need to provide these registers. We do need PC, since there is no external access to the internal IM read address register. PC is a copy of this register.



Function:	Shift:	Skip:	Op:
0: A + B	0: No shift	0: Never	0: ALU: RF[Rw] := F(Ain, Bin), Skip if condition
1: A − B	1: RCY 1	1: ALU < 0	1: RbConst: RF[Rw] := F(Ain, Rb), Skip if condition
2: B + 1	2: RCY 8	2: ALU = 0	2: Input/Output
3: B − 1	3: RCY 16	3: InReady	3: Load: RF[Rw] := DM[RFBout]
4: A and B		4: ALU >= 0	4: Store: DM[RFBout] := RFAout
5: A or B		5: ALU != 0	5: StoreI: IM[RFBout] := RFAout
6: A xor B		6: ~InReady	6: Jump: RF[Rw] := PC + 1, PC := F(Ain, Rb)
7: A and B		7: Always	7: Const: RF[Rw] := IM[26:3]

The instruction set is very simple and regular. All instructions have the same format. Most operations use three register addresses. The ALU instructions do RF[Rw] := function(RF[Ra], RF[Rb]). This seemed easier to explain than a machine that used only one or two register addresses per instruction. Note that the Rb field is 11 bits wide, rather than the 5 bits needed to address RF. This field is used in `Jump` instructions to provide a constant value that

can address the entire instruction memory or provide an 11-bit offset. The `RBconst` instruction uses this field, instead of RFBout, as the B input of the ALU.

Writes to RF[0] are discarded. Because the registers contain zero when the FPGA is configured, r0 will remain zero, and is both a source of zero and a destination in which to discard unneeded ALU results.

Instructions with Ra = 31 replace the ALU's A input with the (zero-extended) PC + 1. This is provided for subroutine linkage.

The `Const` instruction loads RF[Rw] with a 24-bit constant from the instruction. The `In` and `ReadDM` instructions load RF[Rw] with data from an input port or from the data memory. The `Jump` instruction loads RF[Rw] with the incremented program counter. This provides a subroutine return address. `Jump` also uses Rb as the Bin input to the ALU rather than the register file B output, which provides additional flexibility in specifying the target address. In addition, if Rw = 31 in a `Jump` instruction, the PC is loaded with DM[10:0] rather than the ALU output. This saves one instruction in a subroutine return that uses a stack, since it is not necessary to put the return link in a register before jumping to it. Leaf routines (those that do not call other routines) don't need to use a stack, and return by jumping to the Link register.

The ALU, `RbConst`, and input-output instructions conditionally skip the next instruction if the condition selected by the Skip field is true.

The `Store` and `StoreI` instructions write the A output of the RF into the memory location selected by the B output. The `Load` instruction writes RF[Rw] with DM[RF[Rb]].

The machine executes instructions in a single clock cycle, fetching an instruction from the IM location addressed by PC, accessing RF, and doing whatever operation the instruction specifies. At the end of the instruction, the results are written to the register file, and the next instruction is fetched from IM. This is unlike essentially all modern computers, which use pipelining to improve performance, but it is much easier to explain. Pipelining things would make it faster.

Using skips and unconditional jumps is quite powerful and succinct. It was first employed in the 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 between the Tiny Computer and the Nova are:

- There are more registers (32 vs. 4).
- There are three register select fields instead of two.
- The Function field has different meanings.
- The Nova's Carry field has been eliminated.
- The Skip field is different.
- There is no "no load" bit.
- The Nova had a 16-bit data path.

There is no call stack. Programs that need a stack must construct it themselves. The example program below shows how this is done.

The input-output facilities are primitive. Devices are accessed using the IO instruction, which can send RF[Ra] to a device or load RF[Rw] with data from a device. Devices are selected using Rb. A selected device may assert a signal, InReady, that may be tested by a branch condition. There is no support for interrupts, since these add complexity and are difficult to explain.

Size and Speed

In Spartan-3E technology, the machine occupies about 900 4-input LUTs, of which 256 are used for the register file, and 8 block RAMs. This is about 10% of the logic the device contains (although it is 50% of the block RAMs), so there is a lot of room for additional features. It runs at 40 MHz, which is adequate to run nontrivial programs. The Verilog program describing the entire design is a few pages long (see appendix on page 135). As an experiment, I recently built the design for a more modern FPGA, the smallest that Xilinx now sells. It occupies less than 10% of the available logic, demonstrating that Moore's law has not run out yet.

Software

Programs for the TC are written in assembly language, and assembled using the `TCasm` program.

This program takes a single source file and produces a `.coe` file, which the Xilinx tools place into the portion of the bitstream file that specifies the contents of the instruction memory. This entails rebuilding the entire FPGA whenever a program is changed, but since this takes at most a couple of minutes, it has not proven to be a problem in practice.

The assembler is very simple, doing most of its work by looking up textual tokens from the source in a symbol table, and emitting the instruction fragment into the current instruction. The symbol table is populated with definition directives placed at the start of the source file, and by symbols representing labeled statements when these are encountered during assembly. An instruction must occupy a single line of source text, and must be terminated with a semicolon. Any remaining text on the line is ignored. Case is significant. Numbers may be decimal or hex (`0x`*nnn*). Tokens are separated by white space.

The assembler is two-pass. During the first pass, it skips tokens that are undefined. Isolated strings that resolve to fields are placed in `currentValue` with the indicated offset. Isolated numbers are also placed in `currentValue`, using an offset of 3.

Symbol table entries have a `key` and `contents` which is a field containing a `value` and an `offset` in the instruction (the bit position into which the value should be placed). Other built-in reserved words are treated as follows:

`field` ***`string value offset`*** declares a symbol. The symbol has `key=`*`string`* and `contents={`*`value, offset`*`}`.

`rfref` ***`string number`*** defines three symbols, with keys that are concatenations of "`a`," "`b`" and "`w`" with *`string`*. The contents' values are *`number`*, and their offsets are `aoff`, `boff` and `woff`, respectively.

(These constants must be defined as fields before the `rfref`.) For the TC, `aoff=22`, `boff=10`, and `woff=27`. The resulting symbols are used in later instructions to name the register file's `a`, `b`, and `w` instruction fields for register *`number`*.

`mem` ***`number`*** or **`mem`** ***`string`*** (where *`string`* evaluates to a number) makes `M[`*number*`]` the current memory. Memories are numbered `0` to `2`. Usually the *`string`* form will be used, after a preceding `field` definition of the memory name. Token processing continues after the `mem` and its operand.

`loc` ***`number`*** or **`loc`** ***`string`*** (where *`string`* must evaluate to a number) sets the current location in the current memory.

***`string`*`:`** provides a way to label statements. A symbol is defined with `key=`*`string`* and `contents={currentLocation, 11}`. This is a `Rb` constant. Execution of a `Jump loc` instruction substitutes this (11-bit) constant for `RF[Rb]` as the `b` input to the ALU, providing that location `loc` has been labeled. The default value for `Ra` is `0`, and the default function is `add`, so the right thing happens.

When token processing for a line is finished, if any field has been set in `currentValue` then the value is placed into the current location in the current memory, and the current location in that memory is incremented; `currentValue` is then cleared and scanning resumes at the start of the next line.

Because the assembler syntax is so loose, it is easy to write programs that don't work. The usual error is to use the incorrect port variant for named registers. This would usually be worrisome, but we don't expect to write very much assembly code for the machine. To make it a more useful teaching tool, a better assembler would be needed. A simulator would also be useful to try out programs before committing them to the hardware.

The example below shows a very small program, with the machine definition preceding the source code.

```
field aoff  22 0;  Field offsets for rfref.
field boff  11 0;  These must be defined and must not change.
field woff  27 0;

field instruction 0 0; name for instruction memory
field rf 1 0;          name for register file
field data 2 0;        name for data memory

field := 0 0; noise word
field PC 0 0; noise word

field +  0  8; the ‘‘plus’’ function
field -  1  8; the ‘‘minus’’ function
field ++ 2  8; the ‘‘Rb + 1’’ function
field -- 3  8; the ‘‘Rb - 1’’ function
field &  4  8; the ‘‘and’’ function
field |  5  8; the ‘‘or’’ function
field ^  6  8; the ‘‘xor’’ function
field &~ 7  8; the ‘‘and not’’ function

field rcy1  1  6;
field rcy8  2  6;
field rcy16 3  6;

field skn 1  3; skip if ALU < 0
field skz 2  3; skip if ALU = 0
field ski 3  3; skip if InReady
field skge 4 3; skip if ALU >= 0
field sknz 5 3; skip if ALU != 0
field skni 6 3; skip if ~InReady
field skp  7 3; skip always

field RbConst 1 0; Opcodes
field IO      2 0;
field Load    3 0;
field Store   4 0;
field StoreIM 5 0;
field Jump    6 0;
field Call    6 0; as Jump but clarifies intent.  Will specify Rw for the Link.
field Const   7 0;
```

```
mem instruction loc 1;  Make location 1 of instruction memory current.

rfref Trash 0; r0 used for both the trashcan and the source of zero
rfref Zero  0;
rfref Link  1; subroutine linkage register
rfref Stkp 30; stack pointer
rfref PC   31;

; Rb[0] = 0 is In, Rb[0] = 1 is Out
field readRS232Rx  0 11;
field readRS232Tx  2 11;
field writeRS232Tx 3 11;
field writeLEDs    5 11;

; Registers
rfref DelayCount 2;  count this register down to delay
rfref OutValue   3;

start: wStkp := Const 0x7ff; last location in DM
blink: wDelayCount := Const 0xffffff;
       Jump delay  wLink;  subroutine call
       IO writeLEDs aOutValue;
       wOutValue := bOutValue ++;
       Jump blink;

delay:     Store aLink wStkp := bStkp -- ;
delay1:    wDelayCount := bDelayCount -- skz;
           Jump delay1;
ret:       wStkp := bStkp ++ ;
           Load wPC bStkp;

End
```

This program is not very interesting. We have written a few other programs for the system, including a debugging program that communicates with its user using the RS-232 interface. We have not gone as far as providing a compiler for the architecture. Perhaps this should be left as an exercise for the reader.

Extensions

The limited size of the data and instruction memories is the main thing that makes this computer uncompetitive. This could be mitigated by using the memories as caches rather than RAM. The 2 kB BRAM holds 256 blocks of 8 words, which is the usual transfer size for dynamic RAM. We would need to provide I and D tag stores, but this wouldn't be very difficult. The evaluation board contains a 16 MB DDR synchronous dynamic RAM, which could be employed as main storage.

Successors and Conclusions

The Tiny Computer was designed at a time when an interest in using FPGAs as platforms for computer architecture research was growing. In our laboratory, we designed and implemented an example of such a platform, the "BEE3" (Berkeley Emulation Engine version 3). This system contains four Virtex 5 FPGAs, 64 GB of DDR2 memory and a variety of input-output devices. The design was licensed to BEE cube Corporation,[1] which now produces and distributes the systems to researchers throughout the world. Using the BEE3, it is possible for a small team to design and implement serious systems. It has been used by researchers in a number of universities to build systems that are used to explore advanced computer architectures.[2]

While Alan's "handful of gates and flip-flops" was over-optimistic, the Tiny Computer demonstrated that it is possible to build nontrivial systems. Thirty years ago it was common for researchers to build their own computers, program them, and use them in their daily work. The high cost of building silicon chips cut this line of research short. With FPGAs we have seen a resurgence of this sort of activity. In our laboratory we have built a computer system that is used to explore "many-core" architectures, in which a large number of very small processors can be used to build systems of considerable complexity and

[1] `http://www.beecube.com`

[2] Some examples can be found at: `http://www.ramp.eecs.berkeley.edu`

power. The design can be targeted to the BEE3 or to a much less expensive Xilinx development board (XUPv5). On this board, it is possible to build a system with 16 processor cores, a Gigabit Ethernet interface and a DDR2 memory controller. We are using this system as a tool to support our research in computer architecture.

The advent of low-cost FPGA-based boards, coupled with the availability of programming tools to make use of them, makes it possible for students to easily create designs of their own. Perhaps the availability of these devices will enable innovation not only in computer architecture, but in other areas of digital system design. Given the large amount of logic available in modern FPGAs, the high cost of implementing "real" silicon chips need no longer be a barrier to innovation in these areas.

The first computer that I designed that Alan Kay used seriously was the Alto (1973). Alto had a slower clock rate than the TC (170 ns vs. 25 ns). This was the rate at which the machine executed its *micro*-instructions. Real programs, written in real languages such as BCPL and Smalltalk, required several microinstructions to execute each instruction. The Alto had 128 kB of memory and a 2.5 MB disk. The single RAM chip on the least expensive Xilinx development board (in 2007) had six times this amount of storage. The Alto cost $12,000, at a time when $12,000 was a *lot* of money. The Tiny Computer hardware costs $125.

Hardware technology has certainly advanced. Has software? I am still using a program to write this paper that is the lineal descendant of one of the first programs for the Alto: the Bravo text editor. It provided WYSIWYG (what you see is what you get) editing. The Alto had a network (Ethernet), and the first laser printers. It provided a user experience that wasn't much different from the system I'm using today, although today most people have computers, which is quite different.

So we still have a long way to go. Perhaps Alan's most recent attempt to "redefine the personal computer" will help us move forward.

Appendix: Tiny Computer Verilog description

```
‘timescale 1ns / 1ps

module TinyComp(
 input ClockIn,    //50 Mhz board clock
 input Reset, //High true (BTN_SOUTH)
 output [7:0] LED,
 input RxD,
 output TxD
);

wire doSkip;
wire [31:00] WD; //write data to the register file
wire [31:00] RFAout; //register file port A read data
wire [31:00] RFBout; //register file port B read data
reg  [10:0]   PC;
wire [10:0]   PCinc, PCinc2, PCmux;
wire [31:00] ALU;
wire [31:00] ALUresult;
wire [31:00] DM; //the Data memory (1K x 32) output
wire [31:00] IM; //the Instruction memory (1K x 32) output
wire Ph0; //the (buffered) clock
wire Ph0x;
wire testClock;

wire [2:0] Opcode;
wire [4:0] Ra, Rw;
wire [10:0] Rb;
wire Normal, RbConst, IO, Load, Store, StoreI, Jump; //Opcode decodes
wire [2:0] Skip;
wire Skn, Skz, Ski, Skge, Sknz, Skni, Skp;
wire [1:0] Rcy;
wire NoCycle, Rcy1, Rcy8;
wire [2:0] Funct;
wire AplusB, AminusB, Bplus1, Bminus1, AandB, AorB, AxorB;
wire WriteRF;

wire [31:0] Ain, Bin; //ALU inputs

reg  [25:0] testCount;
wire InReady;
wire [31:0] InValue;
reg  [7:0] LEDs;

//--------------- The I/O devices --------------

wire [3:0] IOaddr;  //16 IO devices for now.
wire readRX;
wire charReady;
wire [7:0] RXchar;
wire writeLED;
wire writeTX;
wire TXempty;
wire [7:0] TXchar;

assign IOaddr  = Rb[4:1];  //device addresses are constants.
assign InReady = ~Rb[0] &
```

```
  (((IOaddr == 0) & charReady) |  //read RS232 RX
   ((IOaddr == 1) & TXempty));    //read RS232 TX

assign InValue = (IOaddr == 0) ? {24'b0, RXchar} :  32'b0;
assign TXchar  = RFAout[7:0];
assign readRX  =  ~Rb[0] & (IOaddr == 0) & IO;

assign writeTX =  Rb[0] & (IOaddr == 1) & IO;
assign writeLED = Rb[0] & (IOaddr == 2) & IO;

always @(posedge Ph0) if(writeLED) LEDs <= RFAout[7:0];
assign LED = LEDs;

rs232 user(
 .clock(Ph0),
 .reset(Reset),
 .readRX(readRX),
 .charReady(charReady),
 .RXchar(RXchar),

 .writeTX(writeTX),
 .TXempty(TXempty),
 .TXchar(TXchar),
 .TxD(TxD),
 .RxD(RxD)
 );

 //---------------------- The CPU ------------------------

 always @(posedge testClock)
   if(Reset) testCount <= 0;
        else testCount <= testCount + 1;

  always @(posedge Ph0)
  if(Reset) PC <= 0;
  else PC <= PCmux;

//Opcode fields
assign Rw = IM[31:27];
assign Ra = IM[26:22];
assign Rb = IM[21:11];  //larger than needed to address RF.
assign Funct = IM[10:8];
assign Rcy = IM[7:6];
assign Skip = IM[5:3];
assign Opcode = IM[2:0];

//Opcodes
assign Normal  = Opcode == 0;
assign RbConst = Opcode == 1;
assign IO      = Opcode == 2;
assign Load    = Opcode == 3;
assign Store   = Opcode == 4;
assign StoreI  = Opcode == 5;
assign Jump    = Opcode == 6;
//assign Const   = Opcode == 7;

//Skips
assign Skn =  (Skip == 1);
assign Skz =  (Skip == 2);
```

```
assign Ski =  (Skip == 3);
assign Skge = (Skip == 4);
assign Sknz = (Skip == 5);
assign Skni = (Skip == 6);
assign Skp  = (Skip == 7);

//Cyclic shifts
assign NoCycle = (Rcy == 0);
assign Rcy1 =    (Rcy == 1);
assign Rcy8 =    (Rcy == 2);

//ALU functions
assign AplusB =   Funct == 0;
assign AminusB =  Funct == 1;
assign Bplus1 =   Funct == 2;
assign Bminus1 =  Funct == 3;
assign AandB =    Funct == 4;
assign AorB  =    Funct == 5;
assign AxorB =    Funct == 6;

//The Skip Tester.
assign doSkip =
 (Normal | RbConst | IO) &  //Opcode can skip
   (
    (Skn & ALU[31]) |
    (Skz & (ALU == 0)) |
         (Ski & InReady) |
         (Skge & ~ALU[31]) |
         (Sknz & (ALU != 0)) |
         (Skni & ~InReady) |
         Skp
        );

//The PC-related signals
 assign PCinc =  PC + 1;
 assign PCinc2 = PC + 2;
 assign PCmux =
   Jump ? ALU[10:0] :
   (Load & (Rw == 31)) ? DM[10:0] :  //subroutine return
        doSkip ? PCinc2 :
        PCinc;

//Instantiate the WD multiplexer.
assign WD =
  (Normal | RbConst | Store | StoreI ) ? ALU :
  IO ? InValue:
  Load ?  DM:
  Jump ?  {21'b0, PCinc}:
  {8'b0, IM[26:3]};  // 24-bit constant

assign WriteRF = (Rw != 0); //Writes to r0 are discarded.

//The input multiplexers for the ALU inputs
 assign Ain = (Ra == 31) ? {21'b0, PC} : RFAout;

 assign Bin = ( RbConst | Jump ) ? {21'b0, Rb} : RFBout;

//Instantiate the ALU: An adder/subtractor followed by a shifter
 assign ALUresult =
```

```
   AplusB  ? Ain + Bin :
        AminusB ? Ain - Bin :
        Bplus1  ? Bin + 1 :
        Bminus1 ? Bin - 1 :
   AandB ? Ain & Bin :
        AorB  ? Ain | Bin :
        AxorB ? Ain ^ Bin :
        Ain & ~Bin;  //A and not B

 assign ALU =
   NoCycle ? ALUresult :
   Rcy1 ? {ALUresult[0], ALUresult[31:1]} :
   Rcy8 ? {ALUresult[7:0], ALUresult[31:8]} :
   {ALUresult[15:0], ALUresult[31:16]};

//Instantiate the instruction memory.  A simple dual-port RAM.
 ramx im(
 //the write port
   .clka(Ph0),
        .addra(RFBout[10:0]),
        .wea(StoreI),
        .dina(RFAout),

//the read port
  .clkb(Ph0),
  .addrb(PCmux),
  .doutb(IM)
        );

//Instantiate the data memory. A simple dual-port RAM.
 ramw dm(
 //the write port
   .clka(Ph0),
        .addra(RFBout[10:0]),
        .wea(Store),
        .dina(RFAout),

//the read port
   .clkb(~Ph0),  //use ~Ph0 since we can't read DM until the address (from IM) is ready.
        .addrb(RFBout[10:0]),
        .doutb(DM)  //the read port
        );

//Instantiate the register file.  This has three independent addresses, so two RAMs are needed.
ramz rfA(
  .a(Rw),
  .d(WD), //write port
  .dpra(Ra),
  .clk(Ph0),
  .we(WriteRF),
  .dpo(RFAout) //read port
        );


 ramz rfB(
  .a(Rw),
  .d(WD),
  .dpra(Rb[4:0]),
  .clk(Ph0),
```

```
  .we(WriteRF),
  .dpo(RFBout)  //read port
  );

BUFG ph1Buf(.I(Ph0x),.O(testClock));
BUFG ph0Buf(.I(Ph0x), .O(Ph0)); //Global clock buffer

//The design won't actually run at the 50MHz supplied board clock,
//so we use a Digital Clock Manager block to make Ph0 = 40 MHz.
//This can be ignored, unless you want to change the speed of the design.
DCM_SP #(
.CLKDV_DIVIDE(2.0),
.CLKFX_DIVIDE(10),
.CLKFX_MULTIPLY(8),
.CLKIN_DIVIDE_BY_2("FALSE"),
.CLKIN_PERIOD(20.0),
.CLKOUT_PHASE_SHIFT("NONE"),
.CLK_FEEDBACK("1X"),
.DESKEW_ADJUST("SYSTEM_SYNCHRONOUS"),
.DLL_FREQUENCY_MODE("LOW"),
.DUTY_CYCLE_CORRECTION("TRUE"),
.PHASE_SHIFT(0),
.STARTUP_WAIT("FALSE")

) TCdcm (
.CLK0(),
.CLK180(),
.CLK270(),
.CLK2X(),
.CLK2X180(),
.CLK90(),
.CLKDV(),
.CLKFX(Ph0x),
.CLKFX180(),
.LOCKED(),
.PSDONE(),
.STATUS(),
.CLKFB(),
.CLKIN(ClockIn),
.PSCLK(1'b0),
.PSEN(1'b0),
.PSINCDEC(1'b0),
.RST(Reset)
);

endmodule
```

Chuck Thacker received a B.Sc. in Physics from the University of California at Berkeley in 1968.

He remained at Berkeley with the University's project Genie until leaving for Xerox PARC in 1970. It was at PARC that he met and worked with Alan. Chuck was chief designer of the Alto and co-inventor of the Ethernet local area network.

After some time at Digital Equipment Corporation's Systems Research Center, he joined Microsoft to help establish their laboratory in Cambridge, England. On returning to the U.S. in 1999 he joined the newly-formed Tablet PC group and managed the design of the first prototype machines.

In 2004, Chuck, Butler Lampson, Robert Taylor and Alan were awarded the National Academy of Engineering's Charles Stark Draper prize for the development of the first networked personal computers.

Chuck is currently setting up a computer architecture research group at Microsoft Research in Silicon Valley.