ASIC Implementation of a Two-Stage SMIPSv2 Processor with On-Chip Synchronous RAM Blocks

CS250 Laboratory 3 (Version 092509a)
September 25, 2009
Yunsup Lee

In the second lab assignment, you wrote an RTL model of a two-stage pipelined SMIPSv2 processor using Verilog and synthesized your RTL model. In the third lab assignment, you will substitute magic memory with on-chip synchronous RAM and will use all the tools you have learned so far to simulate, synthesize, place and route your design and finally analyze energy consumption. After producing a preliminary ASIC implementation, you will attempt to optimize your design to make it more energy efficient while meeting your time constraint. The objective of this lab is to introduce SRAM blocks that you might use in your final project, as well as to give you some intuition into how high-level hardware descriptions are transformed into layout.

Before working on the core, you will first make a 32x1024 deep SRAM block out of eight 32x128 SRAM blocks. 32x128 SRAM block is the largest memory building block that comes with the Synopsys 90nm educational standard cell library. After getting the 32x1024 SRAM block right (which includes writing the test harness yourself to test the functionality) you will integrate the SRAM block to your SMIPSv2 core. After producing a working RTL model for the core, you will attempt to optimize your design to increase energy efficiency. The deliverables for this lab are (a) your 32x1024 deep SRAM block implementation checked into SVN, (b) your optimized Verilog source and all of the scripts necessary to completely generate your ASIC implementation and analyze energy consumption checked into SVN, (c) an analytic energy model for your core implementation, and (d) written answers to the critical questions given at the end of this document. The lab assignment is due at the start of class on Tuesday, October 6. You must submit your written answers electronically by adding a directory titled `writeup` to your lab project directory (`lab3/writeup`). Electronic submissions must be in plain text or PDF format.

Before starting this lab, it is recommended that you revisit the Verilog model you wrote in the second lab. Take some time to clean up your code, add comments, and enforce a consistent naming scheme. You will find as you work through this lab assignment that having a more extensive module hierarchy can be very advantageous; initially we will be preserving module boundaries throughout the toolflow which means that you will be able to obtain performance and power results for each module. It will be much more difficult to gain any intuition about the performance or power of a specific assign statement or always block within a module. Thus you might want to consider breaking your design into smaller pieces. For example, if your entire ALU datapath is in one module, you might want to create separate submodules for the adder/subtracter unit, shifter unit, and the logic unit. Unfortunately, preserving the module hierarchy throughout the toolflow means that the VLSI tools will not be able to optimize across module boundaries. If you are concerned about this you can explicitly instruct the VLSI tools to *flatten* a portion of the module hierarchy during the synthesis process. Flattening during synthesis is a much better approach than lumping large amounts of Verilog into a single module yourself.

You are encouraged to discuss your design with others in the class, but you must turn in your own work. The solution for the second lab will be available at `~cs250/lab2.solution` after Monday, September 28. Feel free to reference the code when completing this lab.

Please consult the following tutorials for more information that will help you finish the lab. We strongly encourage you to consult *Tutorial 8: Pushing SRAM Blocks through CS250's Toolflow* to figure out how to instantiate SRAM blocks in your design.

- *Tutorial 2: Bits and Pieces of CS250's Toolflow*
- *Tutorial 3: Build, Run, and Write SMIPS Programs*
- *Tutorial 4: Simulation using Synopsys VCS*
- *Tutorial 5: RTL-to-Gates Synthesis using Synopsys Design Compiler*
- *Tutorial 6: Automatic Placement and Routing using Synopsys IC Compiler*
- *Tutorial 7: Power Analysis using Synopsys VCS and Synopsys PrimeTime PX*
- *Tutorial 8: Pushing SRAM Blocks through CS250's Toolflow*

Figure 6 shows the revised initial template for your SMIPSv2 core. As always please treat it as a suggestion. Notice that the negative edge of the clock triggers reads and writes to the memory.

Block Diagram and Module Interface

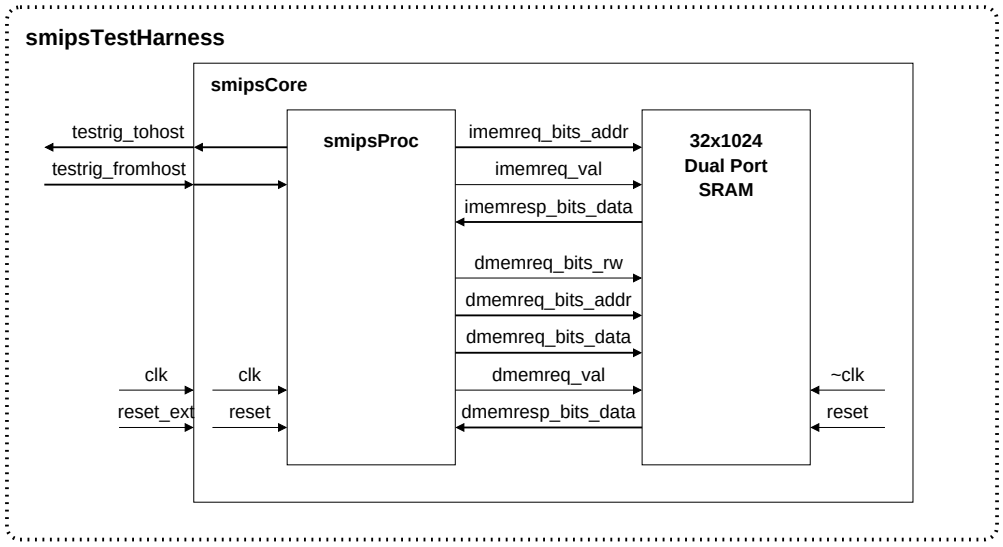

Figure 1: Block diagram for SMIPSv2 Core Test Harness

```
module smipsCore
(
  input clk, reset_ext,

  input  [ 7:0] testrig_fromhost,    // Testrig fromhost port
  output [ 7:0] testrig_tohost,      // Testrig tohost port (must reset to zero)
);
```

Figure 2: Interface for SMIPSv2 Core

Your core should be in a module named `smipsCore` and must have the interface shown in Figure 2. We have provided you with a test harness that will drive the inputs and check the outputs of your design. The `smipsCore` module contains `smipsProc` and `SRAM32x1024`. The interface for `smipsProc` and `SRAM32x1024` is shown in Figure 3.

```
module smipsProc
(
  input clk, reset,

  input  [ 7:0] testrig_fromhost,    // Testrig fromhost port
  output [ 7:0] testrig_tohost,      // Testrig tohost port (must reset to zero)

  output [31:0] imemreq_bits_addr,   // Inst mem port: addr to fetch
  output        imemreq_val,         // Inst mem port: is imem request valid?
  input  [31:0] imemresp_bits_data,  // Inst mem port: returned instruction

  output        dmemreq_bits_rw,     // Data mem port: read or write (r=0/w=1)
  output [31:0] dmemreq_bits_addr,   // Data mem port: read/write address
  output [31:0] dmemreq_bits_data,   // Data mem port: write data
  output        dmemreq_val,         // Data mem port: is dmem request valid?
  input  [31:0] dmemresp_bits_data   // Data mem port: returned read data
);

module SRAM32x1024
(
  input clk, reset,

  output        port0_rdy,           // Port 0: is port ready?
  input         port0_rw,            // Port 0: read or write (r=0/w=1)
  input  [ 9:0] port0_addr,          // Port 0: read/write address
  input  [31:0] port0_input,         // Port 0: write data
  input         port0_input_val,     // Port 0: is request valid?
  output [31:0] port0_output,        // Port 0: returned read data
  output        port0_output_val,    // Port 0: is response valid?

  output        port1_rdy,           // Port 1: is port ready?
  input         port1_rw,            // Port 1: read or write (r=0/w=1)
  input  [ 9:0] port1_addr,          // Port 1: read/write address
  input  [31:0] port1_input,         // Port 1: write data
  input         port1_input_val,     // Port 1: is request valid?
  output [31:0] port1_output,        // Port 1: returned read data
  output        port1_output_val     // Port 1: is response valid?
);
```

Figure 3: Interface for SMIPSv2 Processor and 32x1024 Dual Ported SRAM

Use port 0 for instruction memory and port 1 for data memory. Notice that the response valid signals don't feed into the processor. The processor assumes that whenever it reads or writes data, the data comes back the next cycle or goes through the next cycle.

Test Harness

We are providing a test harness to connect to your core model. The test harness is different to the one you used in previous lab. The test harness loads a SMIPS executable (BVMH format - banked VMH format) into the memory. The provided makefile can build both assembly tests as well as C benchmarks to run on your core. The test harness will clock the simulation until it sees a non-zero value coming back on the `testrig_tohost` register, signifying that your core has completed a test program. The `testrig_tohost` port should be set to zero on reset. A very simple test program is shown in Figure 4. Notice that the reset vector starts from 0x000 rather than 0x1000.

```
# 0x000: Reset vector.
        addiu $2, $0, 1     # Load constant 1 into register r2
        mtc0  $2, $21       # Write tohost register in COP0
loop :  beq   $0, $0, loop  # Loop forever
```

Figure 4: Simple test program

Getting Started

All of the CS250 laboratory assignments should be completed on an EECS instructional machine. Please see the course website for more information on the computing resources available for CS250 students. Once you have logged into an EECS Instructional you will need to setup the CS250 toolflow with the following commands.

```
% source ~cs250/tools/cs250.bashrc
```

Note that to use the toolflow you need to use `bash`, which might not be your default shell on the instructional machine. Login to `update.eecs.berkeley.edu` to change your default shell. Please ask the TA if you have any questions.

You will be using SVN to manage your CS250 laboratory assignments. Please see *Tutorial 1: Using SVN to Manage Source RTL* for more information on how to use SVN. Every student has their own directory in the repository which is not accessible to other students. Assuming your username is `yunsup`, you can checkout your personal SVN directory using the following command.

```
% svn checkout $SVNREPO/yunsup vc
```

To begin the lab you will need to make use of the lab harness located in `~cs250/lab3`. The lab harness provides makefiles, scripts, and the Verilog test harness required to complete the lab. The following commands copy the lab harness into your SVN directory and adds the new project to SVN. Notice that you are making two projects (`v-smipsv2-2stage` and `v-sram32x1024`) for lab3.

```
% cd vc
% mkdir lab3
% svn add lab3
% cd lab3
% mkdir v-sram32x1024
% cd v-sram32x1024
% mkdir trunk branches tags
```

```
% cd trunk
% cp -R ~cs250/lab3/v-sram32x1024/* .
% cd ../..
% mkdir v-smipsv2-2stage
% cd v-smipsv2-2stage
% mkdir trunk branches tags
% cd trunk
% LAB3_ROOT=‘pwd‘
% cp -R ~cs250/lab3/v-smipsv2-2stage/* $LAB3_ROOT
% cd ../..
% svn add *
% svn commit -m "Initial checkin"
% svn update
```

v-sram32x1024 Project

The `lab3/v-sram32x1024/trunk` project directory contains the following subdirectories: `src` contains source Verilog for your 32x1024 SRAM block; and `build` contains automated makefiles and scripts for building the SRAM block.

The `src` directory contains an empty Verilog test harness that needs to be filled as well as an empty Verilog module. The files marked with `(empty)` are the files you need to fill in.

- `SRAM32x1024.v (empty)` - 32x1024 SRAM block
- `SRAM32x1024.t.v (empty)` - Test harness for the 32x1024 SRAM block
- `SRAM32x128.wrap.v` - Wrapper for the SRAM32x128 block

The `build` directory contains the following subdirectories which you will use to test, synthesize, place and route, and analyze energy consumption of your SRAM block.

- `vcs-sim-rtl` - RTL simulation using Synopsys VCS
- `dc-syn` - Synthesis using Synopsys Design Compiler
- `vcs-sim-gl-syn` - Post synthesis gate-level simulation using Synopsys VCS
- `icc-par` - Automatic placement and routing using Synopsys IC Compiler
- `vcs-sim-gl-par` - Post place and route gate-level simulation using Synopsys VCS
- `pt-pwr` - Power analysis using Synopsys PrimeTime PX

Now go ahead and implement a 32x1024 SRAM block out of eight 32x128 SRAM blocks. We recommend you to read *Tutorial 8: Pushing SRAM Blocks through CS250's Toolflow* before you start this part. Do not proceed further until you are done with this project.

v-smipsv2-2stage Project

The resulting `lab3/v-smipsv2-2stage/trunk` project directory looks similar to lab 2's working directory structure: `src` contains your source Verilog; `build` contains automated makefiles and scripts for building your design; `smips-tests` contains local test assembly programs; and `smips-bmarks` contains local C benchmark programs.

The `src` directory contains the Verilog test harness and other Verilog modules you will need in this lab assignment. The files marked with `(empty)` are the files you need to fill in. Copy over your SMIPSv2 implementation from lab 2. Another option is to copy stuff from lab 2 solution.

- `smipsCore.v (empty)` - SMIPS core
- `smipsProc.v (empty)` - SMIPS processor
- `smipsProcCtrl.v (empty)` - Control part of the SMIPS processor
- `smipsProcDpath.v (empty)` - Datapath part of the SMIPS processor
- `smipsInst.v` - SMIPS instruction definition
- `smipsTestHarness_rtl.v` - Test harness for the RTL model
- `smipsTestHarness_gl.v` - Test harness for the gate-level simulation
- `SRAM32x1024.v (copy from the v-sram32x1024 project)` - SRAM block
- `*.v (other files you need from the v-sram32x1024 project)`

The `build` directory contains the following subdirectories which you will use when building your chip.

- `vcs-sim-rtl` - RTL simulation using Synopsys VCS
- `dc-syn` - Synthesis using Synopsys Design Compiler
- `vcs-sim-gl-syn` - Post synthesis gate-level simulation using Synopsys VCS
- `icc-par` - Automatic placement and routing using Synopsys IC Compiler
- `vcs-sim-gl-par` - Post place and route gate-level simulation using Synopsys VCS
- `pt-pwr` - Power analysis using Synopsys PrimeTime PX

Each subdirectory includes its own makefile and additional script files. **If you end up adding more files, you will have to make modification to these script files as you push your design through the toolflow.** Once you have all the tools working you can use the toplevel makefile in the `build` directory to run multiple tools at once. For example, once all the scripts are properly setup you should be able to use the following command to synthesize, place and route, do post place and route gate-level netlist simulation, and power analysis.

```
% cd $LAB3_ROOT/build
% make pt-pwr
```

Now go ahead and copy your 32x1024 SRAM block implementation to the `src` directory. Then change your core design to embed the on-chip synchronous memory. Then you can use the following commands to build local assembly tests and C benchmarks, run them on the SMIPS ISA simulator, build your simulator, run assembly level tests, run benchmarks. You will need to modify the makefile so that it has a listing of all your Verilog source files.

```
% cd $LAB3_ROOT/smips-tests
% make
% make run
% cd $LAB3_ROOT/smips-bmarks
% make
% make run-host
% make run-smips
```

```
% cd $LAB3_ROOT/build/vcs-sim-rtl
% make
% make run-asm-tests
% make run-bmarks-test
```

Guidelines for Lab Grading

Your final lab submission should pass all of the assembly tests and also be able to successfully run the globally installed benchmarks on both RTL simulation and post synthesis and place and route gate-level netlist simulation. The most important metric for lab grading will be energy efficiency (energy/instruction) **using the 3ns time constraint**. Your results on this will be reported. You will also develop your own analytic energy model to predict energy consumption by investigating micro benchmarks and instruction mixes for benchmarks. (see end of document for lab questions).

Lab Hints and Tips

This section contains several hints and tips which should help you in completing the lab assignment.

Tip 1: Always Test Your Processor After Making Modifications

When pushing your processor through the physical toolflow, it is common to make some changes to your RTL and then evaluate their impact on area, power, and performance. Always retest your processor after making changes and before starting the physical toolflow. You can use the `run-asm-tests` make target to quickly verify that your processor is still functionally correct. Keep in mind, that a fast or small processor which is functionally incorrect is worse than a slow or large processor which works!

Tip 2: Replace Non-Synthesizable Verilog

Not all Verilog is synthesizable, so you may receive errors when you first push your processor through Design Compiler. For example, the `smipsInst.v` code which was supplied as part of the first lab included a useful `unpackInst` module. Unfortunately, hierarchical references are not supported by Design Compiler. Instead of using `unpackInst` you can use the `` `define `` macros provided in `smipsInst.v`. To retrieve the rs specifier from an 32 bit instruction wire named `inst` you can use `inst[`INST_RS]`. Another common mistake is to use the Verilog reduction nor operator (`~|`) to implement a two operand nor. VCS will accept (`a ~| b`) but Design Compiler will not. You can change (`a ~| b`) into (`~(a|b)`). Don't forget to retest your design after making any changes!

Tip 3: Reporting Clock Period

As discussed in the tutorials, you need to specify a clock period constraint during both synthesis and place+route. The tools will try and meet this constraint the best they can. If your constraint is too aggressive, the tools will take a very long time to finish. They may not even be able to correctly place+route your design. If your constraint is too conservative, the resulting implementation will be suboptimal.

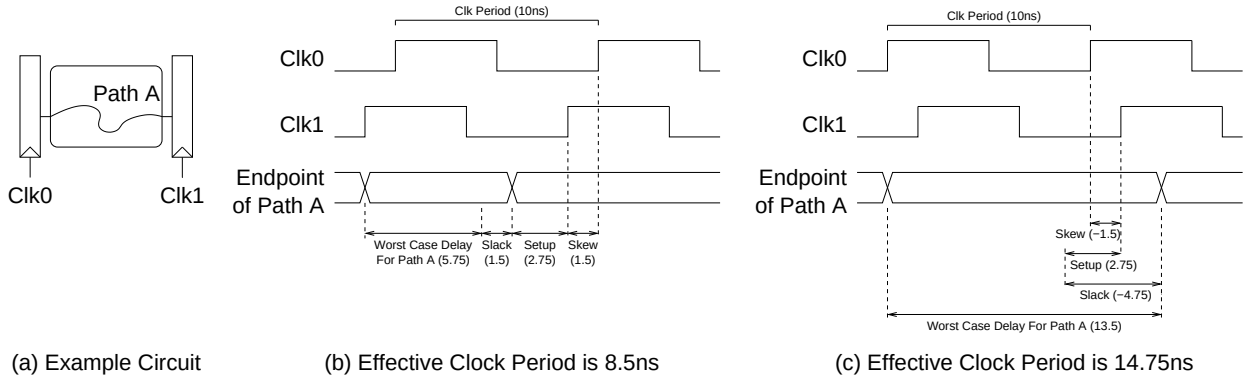

Figure 5: Determining your hardware's effective clock period

Even if your design does not meet the clock period constraint it is still a valid piece of hardware which will operate correctly at some clock period (it is just slower than the desired clock period). If your design does not meet timing the tools will report a *negative slack*. Similarly, a design which makes the timing constraint but does so with a positive slack can run *faster* than the constrained clock period. For this lab we are more concerned about the *effective* clock period of your design as opposed to the clock constraint you set before synthesis. The effective clock period is simply the clock period constraint *minus the worst slack* ($T_{clk} - T_{slack}$). The synthesis and place+route timing reports are all sorted by slack so that the path with the worst slack is listed first. To determine the effective clock period for your design simply choose the smaller of the rising and falling edge slacks. Figure 5 illustrates two examples: one with positive slack and one with negative slack. In this example, our clock period constraint is 10 ns. In Figure 5(b), the post-place+route reports indicate a positive slack value of 1.5 ns and thus the effective clock period is 8.5 ns. In Figure 5(c), the post-place+route reports indicate a negative slack value of 4.75 ns and thus the effective clock period is 14.75 ns. Notice that the effective clock period in Figure 5(c) is *not* equal to the worst case combinational critical path (i.e. 13.5 ns). This is because we must also factor in setup time and clock skew.

Determining the effective clock period when your design does not meet timing can be more tricky when we include latches (as opposed to flip-flops) in our design. Latches impose additional half-cycle constraints. For this lab it is adequate to ignore any half-cycle constraints and focus instead on the full cycle constraints.

Tip 4: Be Careful with the On-Chip Memory

The size of the on-chip memory is 4KB. The last word that can be saved in memory is 0xFFC. The word at address 0x1000 wraps around and will be the same word at address 0x000. To use the limited resources efficiently, assembly test programs and C benchmarks start at 0x000. Be careful of the null pointer since they are not a real null pointer any more - it points to real data. Stack starts at 0xFFC and grows toward 0x000. Keep your program tight, otherwise writing to the stack pointer might end up corrupting instruction data. To support this, you will use a special linker script which can be found at `~cs250/tools/scripts/bvmh.ld`.

Since the on-chip memory is banked, assembly test programs and C benchmarks are saved in BVMH

format. BVMH stands for banked VMH format. For example, in this lab we strip the original memory dump across 8 banks. This is achieved by a different script `objdump2bvmh.py` rather than the original script `objdump2vmh.pl`. When compiled with `objdump2bvmh.py`, the script makes 8 separate files (`*.bvmh.[0-7]`) for each bank.

To manage all these changes, we installed bvmh versions of assembly test programs and C benchmarks to a different location. You can find the installed versions at `~cs250/install/smips-tests-bvmh` and `~cs250/install/smips-bmarks-bvmh`. The source files and the makefile to build these programs can be found at `~cs250/smips-tests-bvmh` and `~cs250/smips-bmarks-bvmh`.

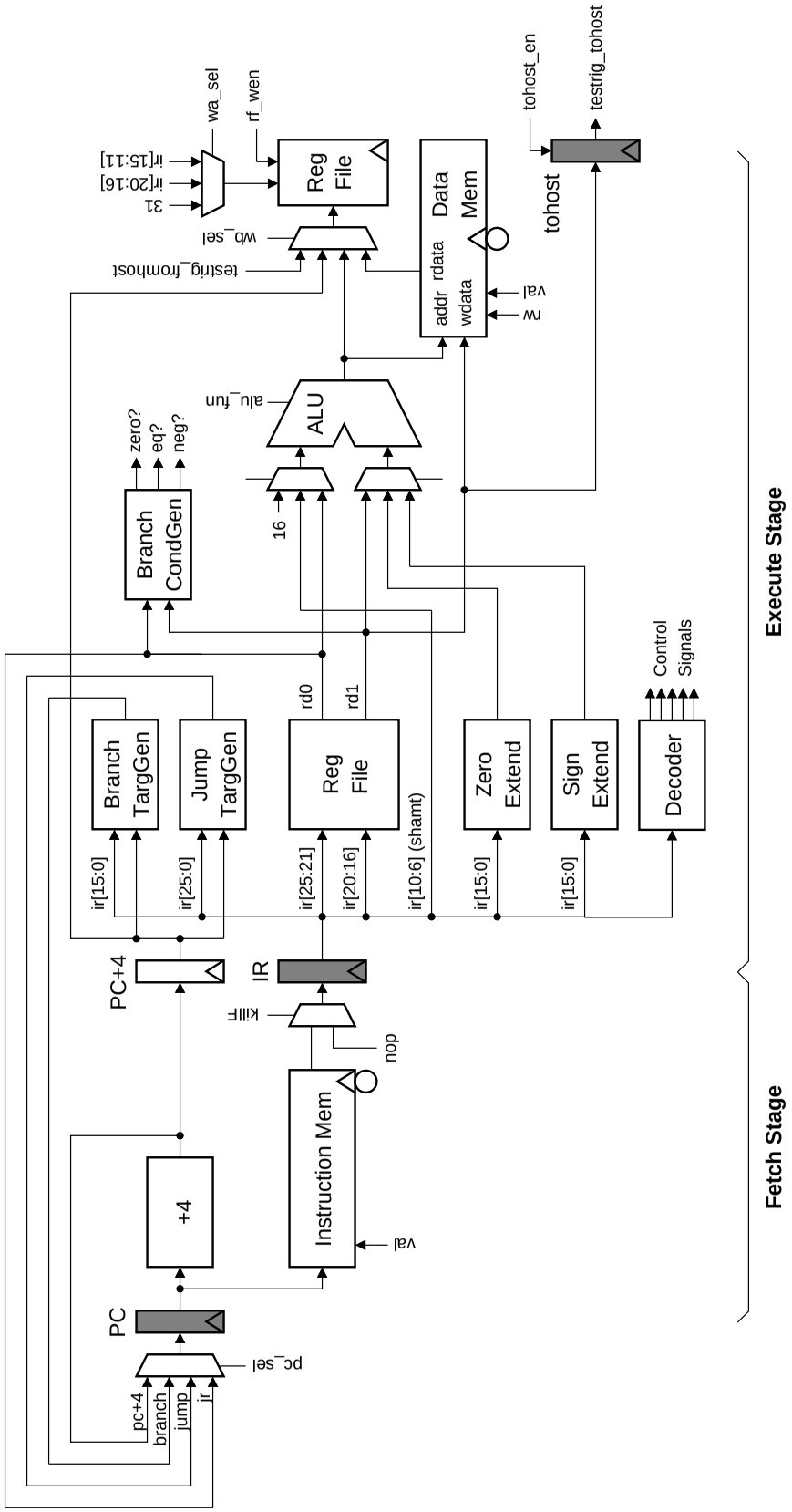

Figure 6: Two-Stage Pipeline for SMIPSv2 Processor. Shaded state elements need to be correctly loaded on reset.

31 26	25 21	20 16	15 11	10 6	5 0	
opcode	rs	rt	rd	shamt	funct	R-type
opcode	rs	rt	immediate			I-type
opcode	target					J-type
Load and Store Instructions						
100011	base	dest	signed offset			LW rt, offset(rs)
101011	base	dest	signed offset			SW rt, offset(rs)
I-Type Computational Instructions						
001001	src	dest	signed immediate			ADDIU rt, rs, signed-imm.
001010	src	dest	signed immediate			SLTI rt, rs, signed-imm.
001011	src	dest	signed immediate			SLTIU rt, rs, signed-imm.
001100	src	dest	zero-ext. immediate			ANDI rt, rs, zero-ext-imm.
001101	src	dest	zero-ext. immediate			ORI rt, rs, zero-ext-imm.
001110	src	dest	zero-ext. immediate			XORI rt, rs, zero-ext-imm.
001111	00000	dest	zero-ext. immediate			LUI rt, zero-ext-imm.
R-Type Computational Instructions						
000000	00000	src	dest	shamt	000000	SLL rd, rt, shamt
000000	00000	src	dest	shamt	000010	SRL rd, rt, shamt
000000	00000	src	dest	shamt	000011	SRA rd, rt, shamt
000000	rshamt	src	dest	00000	000100	SLLV rd, rt, rs
000000	rshamt	src	dest	00000	000110	SRLV rd, rt, rs
000000	rshamt	src	dest	00000	000111	SRAV rd, rt, rs
000000	src1	src2	dest	00000	100001	ADDU rd, rs, rt
000000	src1	src2	dest	00000	100011	SUBU rd, rs, rt
000000	src1	src2	dest	00000	100100	AND rd, rs, rt
000000	src1	src2	dest	00000	100101	OR rd, rs, rt
000000	src1	src2	dest	00000	100110	XOR rd, rs, rt
000000	src1	src2	dest	00000	100111	NOR rd, rs, rt
000000	src1	src2	dest	00000	101010	SLT rd, rs, rt
000000	src1	src2	dest	00000	101011	SLTU rd, rs, rt
Jump and Branch Instructions						
000010	target					J target
000011	target					JAL target
000000	src	00000	00000	00000	001000	JR rs
000000	src	00000	dest	00000	001001	JALR rd, rs
000100	src1	src2	signed offset			BEQ rs, rt, offset
000101	src1	src2	signed offset			BNE rs, rt, offset
000110	src	00000	signed offset			BLEZ rs, offset
000111	src	00000	signed offset			BGTZ rs, offset
000001	src	00000	signed offset			BLTZ rs, offset
000001	src	00001	signed offset			BGEZ rs, offset
System Coprocessor (COP0) Instructions						
010000	00000	dest	cop0src	00000	000000	MFC0 rt, rd
010000	00100	src	cop0dest	00000	000000	MTC0 rt, rd

Figure 7: SMIPSv2 Instruction Set

Critical Thinking Questions

The primary deliverable for this lab assignment is your optimized Verilog source and all of the scripts necessary to completely generate your ASIC implementation checked into SVN. In addition, you should prepare written answers to the following questions and turn them in electronically. There is one bonus question at the end.

Question 1: Take a Screenshot

Take a screenshot of your core and include it in your writeup. You should highlight the register file and the ALU with different colors. Point eight 32x128 SRAM blocks from your design. Do you think pre-placement of the 32x128 SRAM blocks would help minimize core area?

Question 2: Evaluate Your Baseline Core

Push your baseline processor through the physical toolflow and report the following numbers. Use 3ns as your clock period for the numbers below. (Summarize! don't just copy and paste.)

- Post-synthesis area of the register file, processor's datapath (excluding register file), processor's control unit, and 32x1024 SRAM block in um^2 from `*.mapped.area.rpt`
- Post-synthesis total area of core in um^2 from `*.mapped.area.rpt`
- Post-synthesis critical path and corresponding effective clock period in nanoseconds from `*.mapped.timing.rpt`
- Post-place+route area of the register file, processor's datapath (excluding register file), processor's control unit, and 32x1024 SRAM block in um^2 from IC Compiler
- Post-place+route total area of the core in um^2 from IC Compiler
- Post-place+route critical path and corresponding effective clock period in nanoseconds from IC Compiler

Your post-place+route numbers will probably be significantly worse than your post-synthesis numbers. Explain why the place+route tool is reporting more area and a longer clock period than the synthesis tool.

Question 3: Analytic Energy Model for your SMIPSv2 Core

For this question you are to calculate energy/instruction for assembly tests and benchmarks. Notice that you don't need to do the peak power analysis for the benchmarks. It takes too much time to do a time based analysis for benchmark traces. Fill in the table and answer the following questions. You should run two benchmarks you wrote for lab 2 as well.

- Which program has the lowest energy/instruction? The highest energy/instruction? Why?
- Identify the instruction mix for each benchmark program. In this context, instruction mix means the frequency of each instruction that shows up in an execution trace. You can use the trace from the RTL simulation or the ISA simulator.

Program	Average Power	Peak Power	Target Time	Consumed Energy	# Executed Instruction	Energy / Instruction
smipsv1_simple.S						
smipsv1_addiu.S						
smipsv1_bne.S						
smipsv1_lw.S						
smipsv1_sw.S						
smipsv2_addiu.S						
smipsv2_addu.S						
smipsv2_andi.S						
smipsv2_and.S						
smipsv2_beq.S						
smipsv2_bgez.S						
smipsv2_bgtz.S						
smipsv2_blez.S						
smipsv2_bltz.S						
smipsv2_bne.S						
smipsv2_jalr.S						
smipsv2_jal.S						
smipsv2_jr.S						
smipsv2_j.S						
smipsv2_lui.S						
smipsv2_lw.S						
smipsv2_nor.S						
smipsv2_ori.S						
smipsv2_or.S						
smipsv2_simple.S						
smipsv2_sll.S						
smipsv2_sllv.S						
smipsv2_slti.S						
smipsv2_sltiu.S						
smipsv2_slt.S						
smipsv2_sltu.S						
smipsv2_sra.S						
smipsv2_srav.S						
smipsv2_srl.S						
smipsv2_srlv.S						
smipsv2_subu.S						
smipsv2_sw.S						
smipsv2_xori.S						
smipsv2_xor.S						
AVERAGE						
median		N/A				
qsort		N/A				
towers		N/A				
vvadd		N/A				
multiply		N/A				
your benchmark 1		N/A				
your benchmark 2		N/A				

- Can you relate the total energy consumed for each benchmark with the instruction mix that you identified from the previous question and energy/instruction for individual instruction tests?
- Is the model accurate? If not, tell us how you could improve the discrepancy. You might write your own micro benchmarks to improve your analytic energy model. If you write some micro benchmarks go ahead and commit your source code to `smips-tests` or `smips-bmarks`. Run your energy model through the benchmarks and show us how much your energy model improved.

Ideally after this calculation you have your own power/energy model that you can plug into your simulator. Now you don't need to go through all the VLSI tools to get power/energy numbers out!

Question 4: Provide us Feedback

Please provide us feedback about the labs, tutorials, and experience using the VLSI tools.

Read me before you commit!

- For this lab, you don't need to commit any build results. We will build the design from your Verilog source files.
- If you have added Verilog sources codes or changed the name of the Verilog source codes, however, you **must** commit the makefiles you've changed.
- We will checkout the stuff you committed to the `lab3/v-sram32x1024/trunk`, `lab3/v-smipsv2-2stage` and use that for lab grading. Feel free to take advantage of branches and tags. We will also checkout the version which was committed just before 12:30pm on Tuesday, October 6. Use your late days wisely! If you have used your late days, please email TA.
- To summarize, your SVN tree for lab3 should look like the following:

```
/yunsup
  /lab3
    /v-sram32x1024
      /trunk
        /src: COMMIT STUFF YOU HAVE HERE
        /build
          /dc-syn: COMMIT STUFF IF CHANGED
          /icc-par: COMMIT STUFF IF CHANGED
          /pt-pwr: COMMIT STUFF IF CHANGED
          /vcs-sim-gl-par: COMMIT STUFF IF CHANGED
          /vcs-sim-gl-syn: COMMIT STUFF IF CHANGED
          /vcs-sim-rtl: COMMIT STUFF IF CHANGED
      /branches: feel free to use branches!
      /tags: feel free to use tags!
    /v-smipsv2-2stage
      /trunk
        /src: COMMIT STUFF YOU HAVE HERE
        /build
          /dc-syn: COMMIT STUFF IF CHANGED
```

```
        /icc-par: COMMIT STUFF IF CHANGED
        /pt-pwr: COMMIT STUFF IF CHANGED
        /vcs-sim-gl-par: COMMIT STUFF IF CHANGED
        /vcs-sim-gl-syn: COMMIT STUFF IF CHANGED
        /vcs-sim-rtl: COMMIT STUFF IF CHANGED
      /smips-tests: COMMIT STUFF IF CHANGED
      /smips-bmarks: COMMIT STUFF IF CHANGED
    /branches: feel free to use branches!
    /tags: feel free to use tags!
  /writeup: COMMIT STUFF YOU HAVE HERE
```

Acknowledgements

Many people have contributed to versions of this lab over the years. The lab was originally developed for 6.375 Complex Digital Systems course at Massachusetts Institute of Technology by Christopher Batten. Contributors include: Krste Asanović, John Lazzaro, Yunsup Lee, and John Wawrzynek. Versions of this lab have been used in the following courses:

- 6.375 Complex Digital Systems (2005-2009) - Massachusetts Institute of Technology
- CS250 VLSI Systems Design (2009) - University of California at Berkeley