Lab 5: Pipelining an SMIPSv2 Processor: Part I

6.375 Laboratory 5
Assigned: March 4, 2011
Due: March 11, 2011

1 Introduction

In this laboratory assignment and the next you will be provided with an unpipelined three stage SMIPSv2 processor in Bluespec which you must pipeline to acheive good performance. Obtaining good performance requires a solid understanding of how Bluespec schedules rules, something you should be an expert at by the time you complete these labs. For this first lab your task is to produce an elastic pipelined design which functions correctly. The next lab will focus on achieving adequate performance in the pipeline.

This lab handout describes the processor infrastructure, including how to build and run the processor to determine if it functions correctly and how well it performs, advice on how to debug the processor, the initial unpipelined processor design, and detailed steps you should take to successfully pipeline the unpipelined processor.

2 The Processor Infrastructure

A large amount of work has already been done for you in setting up the infrastructure to run, test, evaluate performance, and debug your SMIPSv2 processor in simulation and on the FPGA. This section describes that infrastructure.

Appendix A inclues a reference on the SMIPSv2 instruction set your processor supports.

2.1 Getting Started

Update your local repository with the code from the lab 5 harness. Add the 6.375 course locker, source the setup script, navigate to the directory which contains the `audio/` folder from previous labs, and run

```
$ tar xf /mit/6.375/lab-harnesses/lab5-harness.tar.gz
```

This will create a directory called `smips` with the code for this lab and the next. To save your changes to your local repository, run

```
$ git add smips
$ git commit -m "lab5 initial checkin"
```

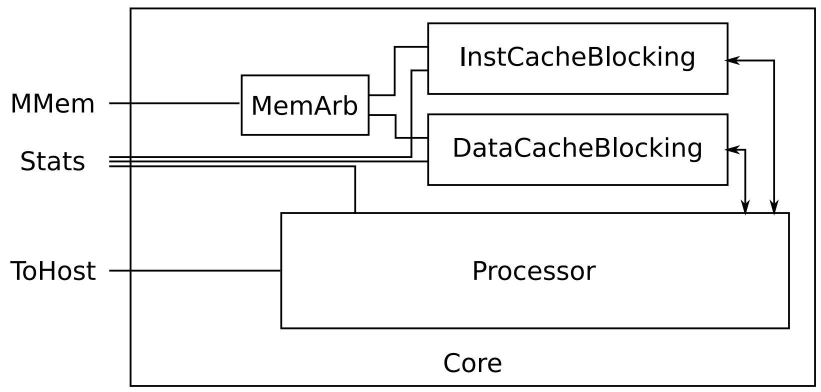


Figure 1: Processor Core

2.2 The Source Code

Figure 1 shows the processor core described by the source code in the `src/` directory. The processor is connected to instruction and data caches. The instruction and data caches interface to the main off-core memory through the memory arbiter. The processor core and both caches collect and expose statistics about the runtime behavior. The processor's `toHost` register used in the MTC0 instruction is also exposed.

The source code implementing the core and all of its components is split into files in the `src/` directory as follows.

`Core.bsv` The top level core. This instantiates the memory arbiter, instruction and data caches, and the processor.

`DataCacheBlocking.bsv` Implementation of the data cache.

`InstCacheBlocking.bsv` Implementation of the instruction cache.

`MemArb.bsv` Implementation of the memory arbiter which arbitrates requests to main memory from the instruction and data caches.

`MemTypes.bsv` Common types related to memory.

`ProcTypes.bsv` Common types related to the processor.

`Processor.bsv` The actual processor. The processor provided is a three stage unpipelined processor described in more detail in section 3. This is where the majority, if not all, of your modifications should be made for this lab.

`SFIFO.bsv` Implementations of searchable FIFOs for use in pipelining the processor.

`Trace.bsv` Definition of the Trace typeclass used for trace output. Tracing is described in more detail in section 2.6.

2.3 The SceMi Setup

Figure 2 shows the SceMi setup for the lab. The SceMiLayer instantiates the core shown in figure 1 and SceMi ports for the core's main memory client, stats, and toHost interface. The SceMiLayer

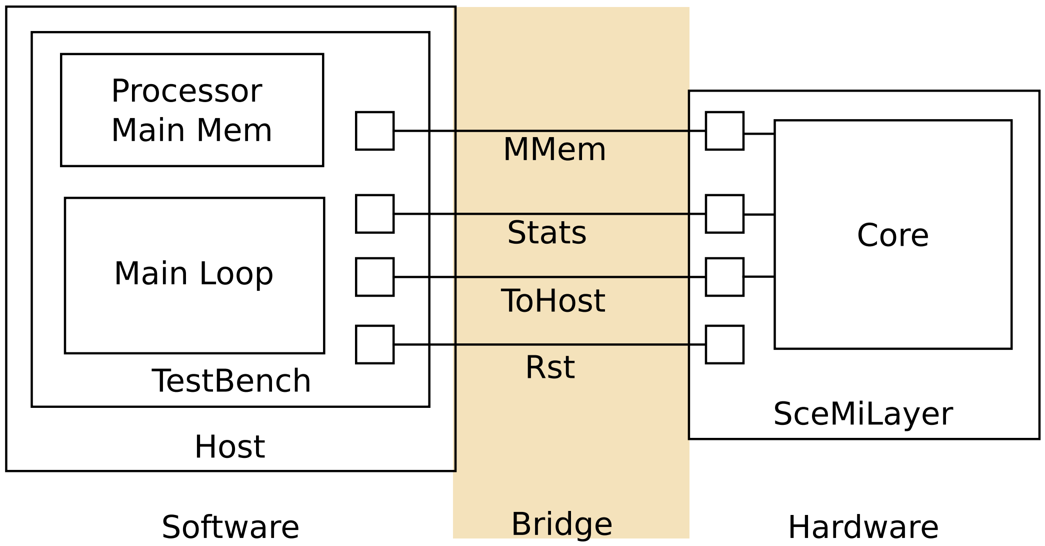


Figure 2: SceMi Setup

also provides a SceMi port for reseting the core from the test bench, allowing multiple programs to be run on the Processor without reprogramming the FPGA.

The core's main memory is implemented in the software test bench running on the host for convenience. Cache misses will result in a memory request being sent accross the SceMi link.

Source code for the SceMiLayer and Bridge are in the `scemi/` directory. The SceMi link goes over a TCP bridge for simulation and a PCIe bridge when running on the actual FPGA.

2.4 Building the Project

The file `sim/project.bld` describes how to build the project using the `build` command which is part of the Bluespec installation. Run `build --doc` for more information on the `build` command. The full project, including hardware and testbench, can be rebuilt from scratch by running the command `build -v` from the `sim/` directory.

The file `sim/sim.bspec` is a Bluespec Workstation project file that can be used to build the project from within the Bluespec Workstation rather than building from the command line. To build the project in the Workstation, run from the `sim/` directory:

```
    bluespec sim.bspec&
```

This opens up the Workstation. From there you can compile and link to generate the two executables `bsim_dut` and `tb`. The executable `bsim_dut` simulates the hardware; `tb` is the test bench.

The `fpga/` directory contains its own `project.bld` and `fpga.bspec` for building the project for the FPGA. Building for the FPGA includes running synthesis, map, and place-and-route, and takes on the order of an hour to complete. When the build has completed, the FPGA can be programmed using the `../tools/program_fpga` command on the FPGA servers. Building for FPGA also creates a `tb` executable for the test bench.

2.5 Using the Test Bench

The test bench is software run on the host processor which interacts with the SMIPSv2 processor over the SceMi link, as shown in figure 2. The test bench loads a program for the SMIPSv2 processor to execute, waits for the processor to send a non-zero value over the `toHost` interface, then collects and displays whatever statistics were gathered in the core when executing the SMIPSv2 program.

The SMIPSv2 programs are specified in Verilog Memory Hex (vmh) format. The `data/` directory contains a number of programs in this format. It is also possible to use the smips-gcc toolchain to compile your own c programs to this format. Ask the TA for more info if you are interested in doing so.

The test bench takes a single command line argument which is the `.vmh` file with the program to run on the SMIPSv2 processor.

To run the test bench, first build the project as described in section 2.4. For simulation the executable `bsim_dut` will be created, which should be running when you start the test bench. For the FPGA you should first program the FPGA with `../tools/program_fpga` on one of the FPGA servers. Then you can call the `tb` executable.

For example, to run the qsort benchmark on the processor in simulation you could use the commands:

```
./bsim_dut 2> trace.txt &
./tb ../data/qsort.smips.vmh
```

The test bench outputs the result of the program and statistics. The SMIPSv2 program could either fail, pass, or timeout, as determined by the value in the toHost register in the SMIPSv2 Processor, which is set by the running SMIPSv2 program.

In simulation the test bench can also be run from the Bluespec Workstation. By default the qsort benchmark is run in the workstation. To change which program to run on the SMIPSv2 processor in the workstation go to `Project->Options`, choose the `Sce-Mi` tab and change the command line arguments to `tb` in the `simulate command` field.

Source code for the test bench is in the `tb/` directory. The `vmh-*` files simulate the main memory for the processor and the `TestBench.*` files implement the main loop and SceMi communications.

2.6 Debugging with Traces

When you run the processor in simulation, the `bsim_dut` outputs to stderr a trace of the execution of the processor. This trace information originates from various uses of the traceTiny() and traceFull() functions in the BSV source code for the core. These functions output a trace tag and some trace data using the Bluespec action `$fdisplay`. This data can be redirected to a file when running the `bsim_dut`. For example, you could run the `bsim_dut` with

```
./bsim_dut 2> trace.txt &
```

which will output the trace data to the file trace.txt once you have run the test bench.

The script `bsv-trace.pl` in the `tools/` directory turns this trace output into a clean text trace format with one cycle per line. The script takes a configuration file as input which describes how to transform the trace output. For example, the following commands can be used to run the addiu

```
processor stage [     icache            ]  [     dcache            ]  [    mem-arb   ]
 pc       [P|X|W] [req|resp|stage|hit/miss]  [req|resp|stage|hit/miss]  [req0|req1|req2]  exInst
...
...
CYC: 1293  pc=         [ | | ]  [    |    |    | ]  [    |    |    | ]  [    |    |    ]
CYC: 1294  pc=         [ | | ]  [    |    |    | ]  [    |    |    | ]  [    |    |100 ]
CYC: 1295  pc=         [ | | ]  [    |    |R   | ]  [    |    |    | ]  [    |    |    ]
CYC: 1296  pc=         [ | | ]  [    |100 |    |h]  [    |    |    | ]  [    |    |    ]
CYC: 1297  pc=         [ |X| ]  [    |    |    | ]  [    |    |    | ]  [    |    |    ]  bne r2, r3, 0x0002
CYC: 1298  pc=00001018 [P| | ]  [100 |    |    | ]  [    |    |    | ]  [    |    |    ]
CYC: 1299  pc=         [ | | ]  [    |    |    |m]  [    |    |    | ]  [    |    |    ]
```

Figure 3: formatted trace

assembly test on the SMIPSv2 processor and produce a clean trace. Part of the trace output is shown in Figure 3.

```
sim$ ./bsim_dut 2> trace.txt & sleep 1 ; ./tb ../data/smipsv1_addiu.S.vmh
sim$ ../tools/bsv-trace.pl ../tools/proc-trace.cfg trace.txt
```

The first column of the clean trace lists the cycle followed by the pc. Next is a character corresponding to the current `stage` or rule which is executing in the processor (`P` for pcGen, `X`, for exec, and `W` for writeback). Then a column corresponding to the state of the instruction cache, followed by a column corresponding to the state of the data cache. The pen-ultimate column displays the state of the memory arbiter while the last shows the instruction being executed. Look at the implementation of each module which outputs traces for a better idea of the meanings of each field. It is relatively simple to add new trace messages, and to extend the perl scripts to format them appropriately. You may find this useful when pipelining your designs as the version provided may not print out all the information you need.

2.7 Assembly Tests

The `data/` directory contains a number of sample programs you can run on your SMIPSv2 processor.

Those files in the `data/` with the `.S.vmh` extension are assembly tests. These are microbenchmarks written in assembly which test specific instructions. Running the assembly tests is a good way to check for errors in your processor implementation and to narrow down which instructions are the source of any problems.

For the assembly tests no statistics are collected. This means the statistics reported for the assembly tests will always be zero, with a bogus IPC. The assembly tests will either pass, timeout, or fail with a specific code corresponding to which subtest within the test failed. To see which subtest corresponds to the return code, look in the `.vmh` file for the test, or the source code for the test, which can be found on the course locker under the `/mit/6.375/install/smips-tests/` directory.

The `tools/` directory contains a couple of scripts you can use for running all of the assembly tests and summarizing the results. The script `tools/sim-asm-tests` runs the assembly tests in simulation. It should be run from the `sim/` directory after the project has been built for simulation. For example:

```
    sim$ ../tools/sim-asm-tests
```

Each of the assembly tests should take no more than a few seconds to complete.

The script `tools/fpga-asm-tests` is used to run all the assembly tests on the FPGA. It should be run from the `fpga/` directory after the FPGA has been programed.

It is highly recommended you rerun all the assembly tests after making any changes to your processor to verify you didn't break anything. Also, run the assembly tests when trying to locate a bug, as they will narrow down which instructions are problematic.

2.8 Benchmarks

Also in the `data/` directory, with the extension `.smips.vmh`, are benchmarks which can be used to evaluate the performance of your processor. Performance is measured in instructions-per-cycle (IPC). The greater the IPC the better. For our pipeline we can never exceed an IPC of 1, but we should be able to get close to it.

Each benchmark consists of a function implemented in C which is used for evaluating performance of the processor. The benchmarks work by first executing the function to load the caches. They then re-execute the function with statistics enabled, and finally verify the result of the function. The function is executed twice so there are no cache misses when statistics are enabled, because cache misses go back to the main memory on the host processor across the SceMi link, which is so slow it will distract from the processor performance that we are really interested in. The trace data output includes information for both runs. You probably want to focus on when the caches are hitting, which happens after the 3rd occurrence of executing `mtc0 r4, cpr10`.

The script `tools/sim-smips-bmarks` runs in simulation each of the benchmarks in turn and reports the stats collected. The benchmarks take longer to run then the assembly tests. Each benchmark may take up to 30 seconds to complete.

The script `tools/fpga-smips-bmarks` runs the benchmarks on the FPGA assuming the FPGA has already been programmed.

3 The Unpipelined Processor

A sketch of the unpipelined processor microarchitecture implemented in `src/Processor.bsv` is given in Figure 4. The processor is split into three mutually exclusive rules corresponding to its three stages: pcGen, exec, and writeback. The `stage` register keeps track of the currently executing stage. For the unpipelined processor only a single instruction is active in the processor at a time. Your task in pipelining the processor will be to allow multiple instructions to be active in the processor at the same time.

The three stages of the unpipelined processor are

pcGen Makes requests to the instruction memory for the next instruction and sets the next stage to exec.

exec Takes the response from the instruction memory, decodes the instruction, reads the register file for operands and executes the instruction. For ALU instructions the register file is updated right away and the next stage set to pcGen. For memory instructions a request is made to data memory and the next stage set to writeback. For branch instructions the pc register is updated appropriately and the next stage set to pcGen.

writeback Takes the response from the data memory and updates the register file for load instructions.

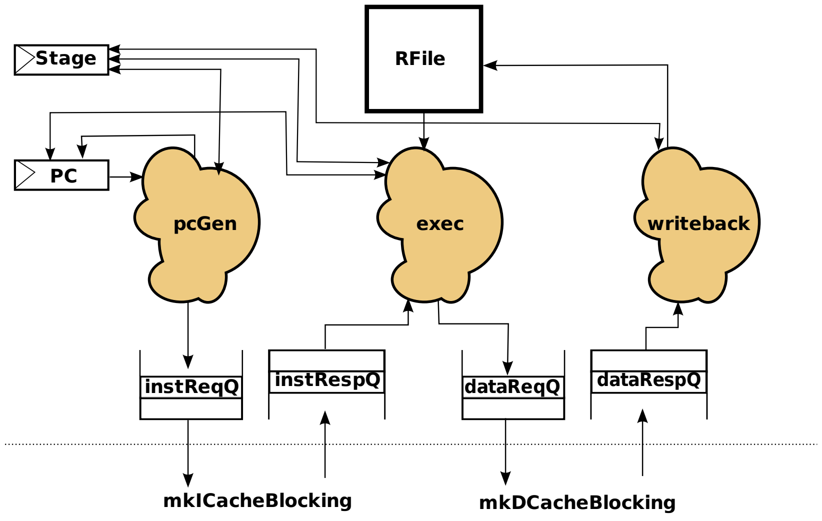


Figure 4: Original Microarchitecture

4 Pipelining the Processor

Pipelining the processor means allowing multiple instructions to be active in the processor at the same time. The three stages pcGen, exec, and writeback should all be firing almost every cycle, each working on a different instruction while maintaining a low critical path through the processor. To achieve this we need to get rid of the stage register in the unpipelined processor and allow the rules to fire independently. In the next lab we will analyze the performance and deal with a number of subtle scheduling issues preventing the rules from actually firing concurrently whenever possible.

This lab focuses on removing the stage register while maintaining the correct functional behavior.

4.1 Moving Register Updates to the Writeback Stage

As the processor is now implemented, the register file is updated sometimes in the exec stage and sometimes in the writeback stage. For our pipelined processor it will be much more convenient if the register file is only updated from a single rule. This means we don't have to worry about the register file being updated out of order. It makes sense to do the update in the writeback rule because we have to wait for load responses before we can do the update in some cases.

Change the exec and writeback stages so the register file is only updated in the writeback rule. You should add a FIFO, call it wbQ, between the exec and writeback stages which holds the register destination and value for ALU type instructions and the register destination for LOAD type instructions. All of the ALU instructions in exec should go to the writeback stage instead of pcGen. Defining your own data type for storing this information in the wbQ is a good idea. Later we'll refer to this data type as a WBResult.

To ease the transition we have encapsulated all calls to `rf.wr` for arithmetic instructions in an Action function called `wba`. You will want to change the implementation of this function.

After making your changes, `rf.wr` should only be called from the writeback rule, and not from the exec rule. Run all of the assembly tests and benchmarks to verify they pass before moving on to the next step. You may also want to check your revisions into your local git repository once you get this working.

4.2 Letting Writeback Run Independently

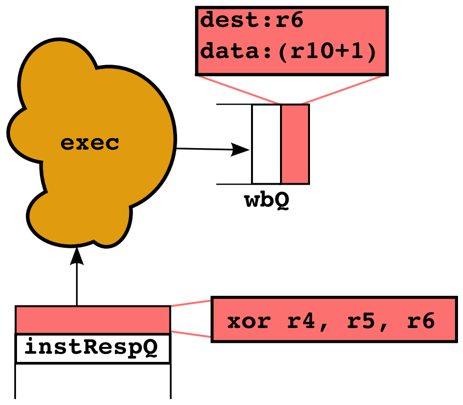
Now that all the register file updates are performed in the writeback rule, we can separate the writeback rule from pcGen and exec, allowing it to run currently with the pcGen and exec rules, one step closer toward a fully pipeliend processor.

With this change, the `stage` register will alternate back and forth between pcGen and exec. The writeback rule will no longer have any explicit guard, firing whenever there is anything for it to write back to the register file.

We have to be careful when letting the exec and writeback rules fire separately that we don't break the functionality of the processor. Is it possible the exec rule could read a register file before the writeback rule had a chance to update it? Consider the following program.

```
A) addiu r6, r10, 1
B) xor   r4, r5, r6
```

The processor will eventually reach the following state:



Instruction B cannot be executed because it reads `r6`, and instruction A is responsible (and has not yet) written the new value back to the register file. This means we need to stall the execution of instruction B until the instruction A writes the register file.

We will detect these Read-After-Write (RAW) hazards by making wbQ a `SFIFO`, a searchable FIFO with the following interface:

```
interface SFIFO#(type any_T, type search_T);
    //Standard FIFO methods
    method Action enq(any_T data);
    method Action deq();
    method any_T first();
    method Action clear();
    //New SFIFO methods
    method Bool find(search_T searchVal);
    method Bool find2(search_T searchVal);
endinterface
```

SFIFO is just like a normal FIFO with two extra methods: `find()` and `find2()`. These methods take a value to search for within the SFIFO, returning True if the given parameter is present in the FIFO, and False otherwise. `find()` and `find2()` have no implicit condition — they are always ready — they will simply return False if the FIFO is empty.

Why does SFIFO include two methods `find()` and `find2()`? This so you can search it twice for instructions that have two operands. For instance, in the above example the execute rule can check if `r5` is in the `writebackQ` using `find()`, and `r6` using `find2()`. However if the instruction had been `xori` instead of `xor`, it just would have used `find()` because `xori` just has one register argument.

You must define what it means to search the SFIFO by writing a function. This function should take an Rindx (the thing we're looking for) and a WBResult (the elements through which we are searching). The function should return true if the search value is "found" in the given FIFO element.

```
function Bool findf(Rindx searchval, WBResult val);
  //You write this
endfunction
```

When you instantiate the SFIFO, you should pass in the appropriate types and find function. What does it mean to pass in a function to a hardware module in Bluespec? Essentially it means that when the compiler instantiates the module it will do so with the combinational logic you provide. Think of the SFIFO as a black box — a black box with a hole in it. The function you provide fills that hole.

Thus the types of the writeback queue is as follows:

```
//Searchable for stall signal
SFIFO#(WBResult, Rindx)  wbQ  <- mkSFIFO(findf);
```

It would be good to encapsulate the stall signal inside a function which reads the wbQ and instruction to execute and returns True if the execute stage must stall and False otherwise. This function should be added to the explicit guard of the exec rule.

Make these changes to your processor. The stage register should never enter Writeback, the writeback rule should have no explicit guard, and the exec rule should stall when appropriate to avoid Read-After-Write hazards. After you've made these changes rerun the assembly tests and benchmarks to verify you haven't broken anything, then check your latest work into your local git repository.

4.3 Letting PCGen and Exec Run Independently

We are now at the point where we can remove the stage register completely, freeing the pcGen and exec stages to run independently.

The trouble is, pcGen depends on the result of exec to know what the next pc should be. We can get around this by having the pcGen stage guess what the next pc should be. A good guess for the next pc would be pc+4. We will improve this in the next lab by implementing a smarter branch predictor, but for now pc+4 will work fine as our branch prediction algorithm.

To maintain functional correctness in light of the pcGen stage guessing the next pc we need a way to detect when pcGen's guess was wrong and undo whatever work pcGen started incorrectly.

There are now two notions of the current pc. There is the speculative pc pcGen runs off of, and the correct pc exec runs from. The pcGen stage should send information via a FIFO, call it pcQ, to the exec stage so exec is able to detect when the next pc predicted by the pcGen stage is different from the next pc the exec stage calculates is the correct next pc.

When pcGen messes up, there will be bad instruction requests made to memory which need to be killed. A simple way to kill these wrong path instructions is to associate an epoch with every instruction. An epoch is a conceptual grouping of all instructions in between branch mispredictions. The epoch is incremented when a misprediction is detected. Only instructions from the current epoch are executed, instructions from the old epoch are killed. You may wish to introduce a new rule especially for killing instructions with an old epoch.

When you have made these changes your processor should no longer have or refer to the stage register. Rerun the assembly tests and benchmarks to verify nothing has broken. Check in your latest work to your local repository, then push that to your repository on the course locker to submit the lab. For example:

```
smips$ git add -u .
smips$ git commit -m "Lab 5 submission"
smips$ git push
```

A SMIPSv2 Instruction Set

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 5: SMIPSv2 Instruction Set