Before we move on to updating our basic cpu verilog code let us talk a little more about the four basic units of a cpu. These parts are typically called the Datapath, Function Unit or ALU, Memory/Register File and the Control Unit. Any cpu must at least have these four basic blocks.

Memory / Register File

Internal memory for cpus are normally called ***registers*** rather than memory. The collection of registers being called the Register File. Most cpus contain very limited amounts of internal memory or registers and the contents of these registers are loaded to and from main external memory or intermediate caches quite often. Some cpus like the intel x86 models have functionaly specific registers like AX,BX,ES,BP,SS,DS (to name a few), while other cpus simply have a homogeneous register file containing 0 to n register. RISC architecture chooses the latter method of having ahomogeneous register file, so we shall to in our design.

Function Unit or Arithmetic Logic Unit (ALU)

This cpu block performs the calculations or data transformations for each execution cycle. All cpu operations possible are difined and implemented in this cpu block. Care is taken to optimize this section of the cpu as it tends to have the longest delays in logic. Furthermore, some operations may activate multiple operations in the ALU such as inverting and incrementing a register location (2's complement encoding of negative numbers) or decrement and test for zero/non-zero. Combining all operations into a single ALU allow for larger instruction sets using less logic.

Datapath

The datapath is the piping, so to speak, between the various blocks of the cpu. The datapath carries the data between the cpus internal memory and the ALU and back to memory again. It may also channel register memory into and out of the cpu to external memory. While externally a cpu may only have a single address/data bus, internally a cpu may have multiple seperate buses. In our RISC cpu design,we have a datapath for each of the ALU operad inputs and another to carry the ALU output back to register memory. In a later chapter, when we integrate external memory we will interface to these datapaths arbitrating the data between external memory and the register file.

Control Unit

This module sequences each of the other 3 modules, decoding the next instruction and then controlling the flow of execution between the memory, ALU and back to memory. There are primarily two implementation methods to building a Control Unit. A control unit can be designed

from combinational logic that given the clock cycle and decoded instruction generates outputs to each of the other blocks. Secondly, some cpu's employ a microsequencer which is similar to a very small and simple cpu itself, with a simple instruction set, that uses these micro-instructions to generate outputs to the other blocks. A microsequencer is found in more complex cpus such as those used in PC desktops and servers. Our cpu will contain a control unit of the first kind, using combinational logic.

Bussing Data

If you know anything of how computers work you are aware that any computer has an address and a data bus. This bus carries information between the memory, hard drive controllers and other peripherals. Inside a cpu, busses exist to carry the information from memory (or cache) to the function unit and place the result back into memory. These busses, and there is often many, are not the same bus as the cpu external bus and in fact is normally only connected by some sort of arbitrator (gateway).CPU busses are (normally) not accessible outside the cpu package. Often, and as in our cpu, the buses will be unidirectional and data will flow from memory to our function unit and back into memory. Our function unit will have 1 or 2 operands, thus to load two operands from memory in a single clock cycle we will require 3 buses in total, named A, B, and D. The buses A+B will be routed into our functional unit as left and right operands. Bus D will be the result from the functional unit and will perform a write back to memory.Our CPU will be a RISC type CPU.

Branching in our CPU

In this chapter we want to add branching instructions to create a cpu that can run real programs! Without looping our cpu remains a basic calculator. We will add five branching instructions:

- **BRA** - Branch to location in REGFILE[RB]
- **BRANZ** - Branch to location in REGFILE[RB] if REGFILE[RA] is non-zero
- **BRAL** - Branch to literal location RB
- **BRALNZ** - Branch to literal location RB if REGFILE[RA] is non-zero
- **CALL** - Call a subroutine at literal location RB, store return location in RD

Two instructions branch to instruction locations referenced in a register (BRA) or a literal location (BRAL) specified by RB. Each of these instructions have a conditional version, BRANZ & BRALNZ respectively, that branch only if register referenced in RA is non-zero.

A fifth instruction calls a subroutine and stores the return address in register RD. This allows simple coding of functions and subroutines in our assembler or compiler. The compiler would be responsible for managing stack frames properly.

Easily add more mnemonics, such as a "Decrement RA and jump if non-zero". I have left this excercise to you. As well as the very useful subroutine mnemonics that store PC into a register and jump.

As branch instructions don't store back into the destination register a new Control Unit state was added that skips the write-back cycle. We could simply have the execute cycle move to s_fetch as the next state but then branch operations would take only 3 cycles where all others take 4. For deterministic timing this behaviour was choosen. Later, this will help us in pipelining our processor.

Testbench

The testbench remains the same as Chapter 1.

Chapter 2 - Branching (tgz)

Chapter 2 - Branching (zip)