TinyMIPS

This document briefly describes TinyMIPS, a tiny subset of the Microprogrammed Instruction Processing Simulator, which has been implemented as GateSim logic for easy modification and extension. See the Software page on the CD for instructions on installing the GateSim simulator. For a somewhat more complete specification of the MIPS architecture, see the Appendix A on the CD accompanying *Computer Organization and Design*.

TinyMIPS is more or less compatible with the Spim simulator. MIPS assembly language programs can be assembled in Spim, then the result saved as a log file that TinyMIPS can read and decode. See the Software page on the CD for instructions on installing the Spim simulator.

TinyMIPS Limitations

TinyMIPS has only a shell of the floating-point coprocessor instructions, none of the multiply and divide operations, no I/O or system call operations, no exceptions, no inclusive-OR and only one single-bit right shift. It can only load and store full words in memory. Program and data storage are limited to 1024 words (4K bytes) each. Note that memory is word-addressed only.

The following MIPS instructions should work correctly in TinyMIPS:

Move	ALU	Jump
lui	addu	jump
lw	addui	jr
sw	subu	jal
mfhi	and	jalr
mthi	andi	beq
mflo	xor	bne
mtlo	xori	blez
	slt	bgtz
	sltu	bltz
	slti	bgez
	sltiu	

The three signed arithmetic operations, `add`, `addi`, and `sub`, work exactly the same as their unsigned cousins. The only effective difference between them and MIPS is that they cannot cause an exception in case of overflow.

The pseudo-instruction `li` (load immediate) sometimes assembles to the machine operation `ori`, which is not implemented in TinyMIPS. The `ori` instruction executes as `xori`, which gives the same results in the `li` pseudo-instruction sequence, and wrong results otherwise. Similarly, the `nop` (no operation) pseudo-instruction is actually a special case of the MIPS `sll` instruction, shifting register 0 left by 0 places; in TinyMIPS all `sll` instructions are no-operations. The TinyMIPS `syscall` instruction is also implemented as a no-operation. These adaptations were necessary to accommodate the code generated by the SPIM assembler.

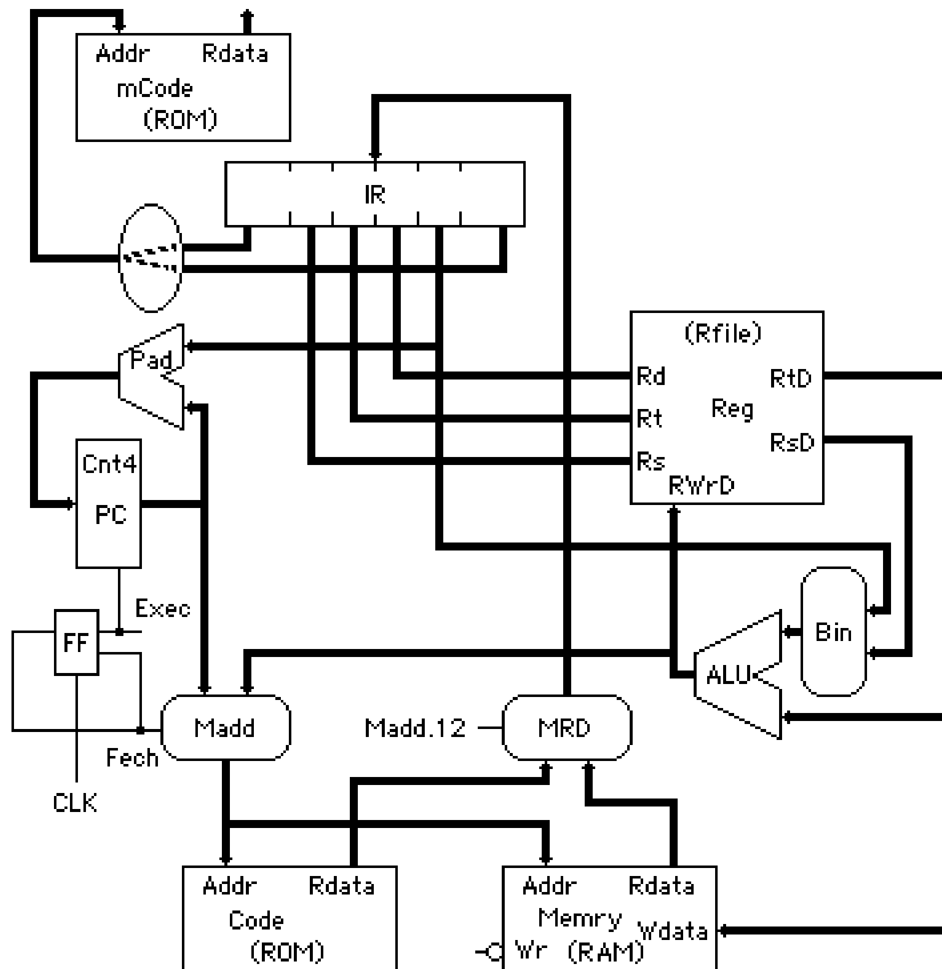
The `sra` (shift right arithmetic) instruction is the only shift implemented in TinyMIPS, and it only shifts by one place; any attempt to shift by more than one place will be ignored.

Using TinyMIPS

The TinyMIPS file (TinyMIPS.txt) has no MIPS executable code in it; you must add that yourself. After you have a program working in the Spim simulator, you can save a log of your session then direct GateSim to that file as input. GateSim will extract your assembled code from the log file and insert it into a copy of the TinyMIPS file, and execute the result. The output log file from GateSim will also contain the DATA lines extracted from your log file; you can cut them out and splice them into any other copy of the TinyMIPS file for further modification.

The TinyMIPS Architecture

It is recommended that you open the TinyMIPS.txt file in a separate window, for reference in the following discussion. The simplified diagram below shows how the circuit is wired up. Note that most of the control logic is not shown.



The front of the gate specification defines four memories: The first ROM and RAM are logically combined to be the main system memory; they are separate circuits mostly because the SPIM assembler puts code and data into different addresses. These are followed by two ROMs, which are effectively programmable logic arrays (PLAs) that serve to decode the MIPS instructions. (The floating-point ROM, FDROM, is not shown in this diagram.) The dots in the binary ROM words are used as "don't-care" fillers (recognized as `0`s by GateSim). Each location in a ROM is one decoded instruction, accessed by the bits of the top and bottom 6-bit segments of the Instruction Register (IR), that should execute in one cycle. Each bit (or sometimes a cluster of bits) defines one control operation, such as enabling a conditional or unconditional jump, selecting the jump address, and so on. Almost all the floating-point operations have a single opcode (`0x11`) that is further decoded in the second PLA from IR bits not otherwise decoded. However, only the register transfer and conditional branches are implemented. Floating-point arithmetic is left as an exercise for the student.

Next in this specification is the Rfile macro defining 31 1-bit three-port registers together with the logic for reading and writing them. This macro is used for both the general register file and for the floating-point registers immediately following it. In order not to replicate the register selection decoder for writing the destination register, there are 31 separate decoded write control lines (register 0 cannot be written, and always reads out as `0`).

Next is the Program Counter (PC) and the logic for incrementing it and selecting conditional jumps. The PC is only 12 bits, built up from three 4-bit counters. The low-order two bits are not implemented; they are always

assumed to be `00`. The PC is always incremented at the end of the Fetch cycle and then conditionally reloaded at the end of the Execute cycle if a jump occurs. Conditional branches have relative addresses, so there is a separate adder to add the offset from the IR to the unincremented PC value saved from before the PC was incremented.

The TinyMIPS specification has places in which to insert multiply and divide instructions, but they are left as an exercise for the student. The Hi and Lo result registers are already provided and and are functional for transferring to/from the general registers.

The ALU is only capable of three operations: ADD (with the carry turned off), AND and XOR. These are implemented in a 1-bit macro, which takes A- and B-inputs, the carry from the next lower bit and a couple of control bits to select the operation. This version of the ALU is ripple-carry only, which takes two gate delays for every bit; the clock has been slowed down to 100 gate-delays per cycle to accommodate it. A fast look-ahead carry generator can be added as an exercise.

Toward the end of the TinyMIPS specification file are the random logic control signals. Execution timing proceeds in two (rather long) cycles, Fetch then Execute, generated by a flip-flop toggle (shown in the lower left corner of the diagram above). This toggle bit controls whether the memory address comes from the PC or the offset+register calculation in the ALU for loads and stores. It also enables the execution of the register write-back and conditional branch logic in those instructions that do those things. An exercise could alter the timing to lengthen the slower instructions and speed up the faster ones, by replacing this flip-flop with a state counter. It would be completely eliminated for pipelining or other structural changes.

The end of the file is devoted to trace specification. There are a number of trace specifications that are commented out; these can be turned on to see some of the machine operation in more detail.