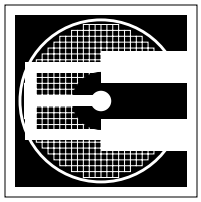


The Pipelined RiSC-16

ENEE 446: Digital Computer Design, Fall 2000
Prof. Bruce Jacob

This paper describes a pipelined implementation of the 16-bit Ridiculously Simple Computer (RiSC-16), a teaching ISA that is based on the Little Computer (LC-896) developed by Peter Chen at the University of Michigan.

1. RiSC-16 Instruction Set

The RiSC-16 is an 8-register, 16-bit computer. All addresses are shortword-addresses (i.e. address 0 corresponds to the first two bytes of main memory, address 1 corresponds to the second two bytes of main memory, etc.). Like the MIPS instruction-set architecture, by hardware convention, register 0 will always contain the value 0. The machine enforces this: reads to register 0 always return 0, irrespective of what has been written there. The RiSC-16 is very simple, but it is general enough to solve complex problems. There are three machine-code instruction formats and a total of 8 instructions. The instruction-set is given in the following table.

Assembly-Code Format		Meaning
`add`	`regA, regB, regC`	R[regA] <- R[regB] + R[regC]
`addi`	`regA, regB, immed`	R[regA] <- R[regB] + immed
`nand`	`regA, regB, regC`	R[regA] <- ~(R[regB] & R[regC])
`lui`	`regA, immed`	R[regA] <- immed & 0xffc0
`sw`	`regA, regB, immed`	R[regA] -> Mem[R[regB] + immed]
`lw`	`regA, regB, immed`	R[regA] <- Mem[R[regB] + immed]

`beq`	`regA, regB, immed`	if (R[regA] == R[regB]) {
		PC <- PC + 1 + immed
		(if label, PC <- label)
		}

`jalr`	`regA, regB`	PC <- R[regB], R[regA] <- PC + 1
PSEUDO-INSTRUCTIONS:		
`nop`		do nothing
`halt`		stop machine & print state
`lli`	`regA, immed`	R[regA] <- R[regA] + (immed & 0x3f)
`movi`	`regA, immed`	R[regA] <- immed
`.fill`	`immed`	initialized data with value *immed*
`.space`	`immed`	zero-filled data array of size *immed*

The instruction-set is described in more detail (including machine-code formats) in *The RiSC-16 Instruction-Set Architecture*.

2. Pipelined Implementation

A non-pipelined implementation of the RiSC-16 is described in the document *RiSC-16: Sequential Implementation*. The document shows the control flow and data flow for each instruction, as well as the final hardware implementation that changes its dataflow based on the instruction opcode. In that example, the entire instruction must be executed before the next clock, at which point the results of the instruction are latched in the register file or data memory. This results in a relatively long clock period.

The computer market is not fond of slow clocks, however. Increased clock speeds are possible as the amount of logic between successive latches is decreased. If execution is sliced up into smaller sub-tasks, the clock can run as fast as the longest sub-task. Theoretically, a pipeline of N stages should run with a clock that is N times faster than a sequential implementation. For many reasons, this theoretical limit is never reached, due to latch overhead, sub-tasks of unequal length, etc. Nonetheless, extremely fast clock rates are possible. Slicing up the instruction execution this way is called *pipelining*, and it is exploited to great degree in nearly every aspect of modern computer design, from the processor core to the DRAM subsystem, to the overlapping of transactions on memory and I/O buses, etc.

The RiSC-16 pipeline is shown in Fig. 1 on the next page. It is similar to the 5-stage DLX/MIPS pipeline that is described in both *Hennessy & Patterson* and *Patterson & Hennessy*, and it fixes a few minor oversights, such as lack of forwarding to store data, lack of forwarding to comparison logic in decode implementing the 1-instruction delay slot, etc. This pipeline adds in forwarding for store data and eliminates branch delay slots. As in the DLX/MIPS, branches are predicted not taken, though implementations of more sophisticated branch prediction are certainly possible.

In the figure, shaded boxes represent clocked registers; thick lines represent 16-bit buses; thin lines represent smaller data paths; and dotted lines represent control paths. The figure illustrates how pipelining is achieved: the sub-tasks into which instruction execution has been divided are instruction fetch, instruction decode, instruction execute, memory access, and register-file writeback. Each of these sub-tasks, which is executed by dedicated hardware called a *pipeline stage*, produces intermediate results that must be stored before an instruction may move on to the next stage. By breaking up execution into smaller sub-tasks, it is possible to overlap the different sub-tasks of several different instructions simultaneously. If the intermediate results of the various sub-tasks are not stored, they would be lost, as during the next cycle another instruction would be using the same hardware for its own sub-task. For instance, after an instruction is fetched, it is necessary to store the fetched instruction somewhere, because the output of the instruction memory will be different on the following cycle—the fetch stage will be fetching a completely different instruction.

The storage locations for the intermediate results are called *pipeline registers*, and the figure illustrates their contents. It is common to label a pipeline register with the two stages that it divides. For example, the pipeline register that divides the instruction fetch (IF) and instruction decode (ID) stages is called the *IF/ID register*; the register that divides the instruction execute (EX) and memory-access (MEM) stages is called the *EX/MEM register*; and the register at the end of the register-file writeback (WB) stage could be called the *WB/END register.*

Note that neither the WB/END register nor the data-forwarding path it supports is present in the DLX/MIPS architecture described by Hennessy & Patterson. The DLX/MIPS assumes a half-cycle register-file access, so that the writeback stage completes in the first half of the cycle and the register-file read component of the decode stage happens in the second half of the cycle. This allows data to be forwarded from the writeback stage to the decode stage directly. Otherwise such

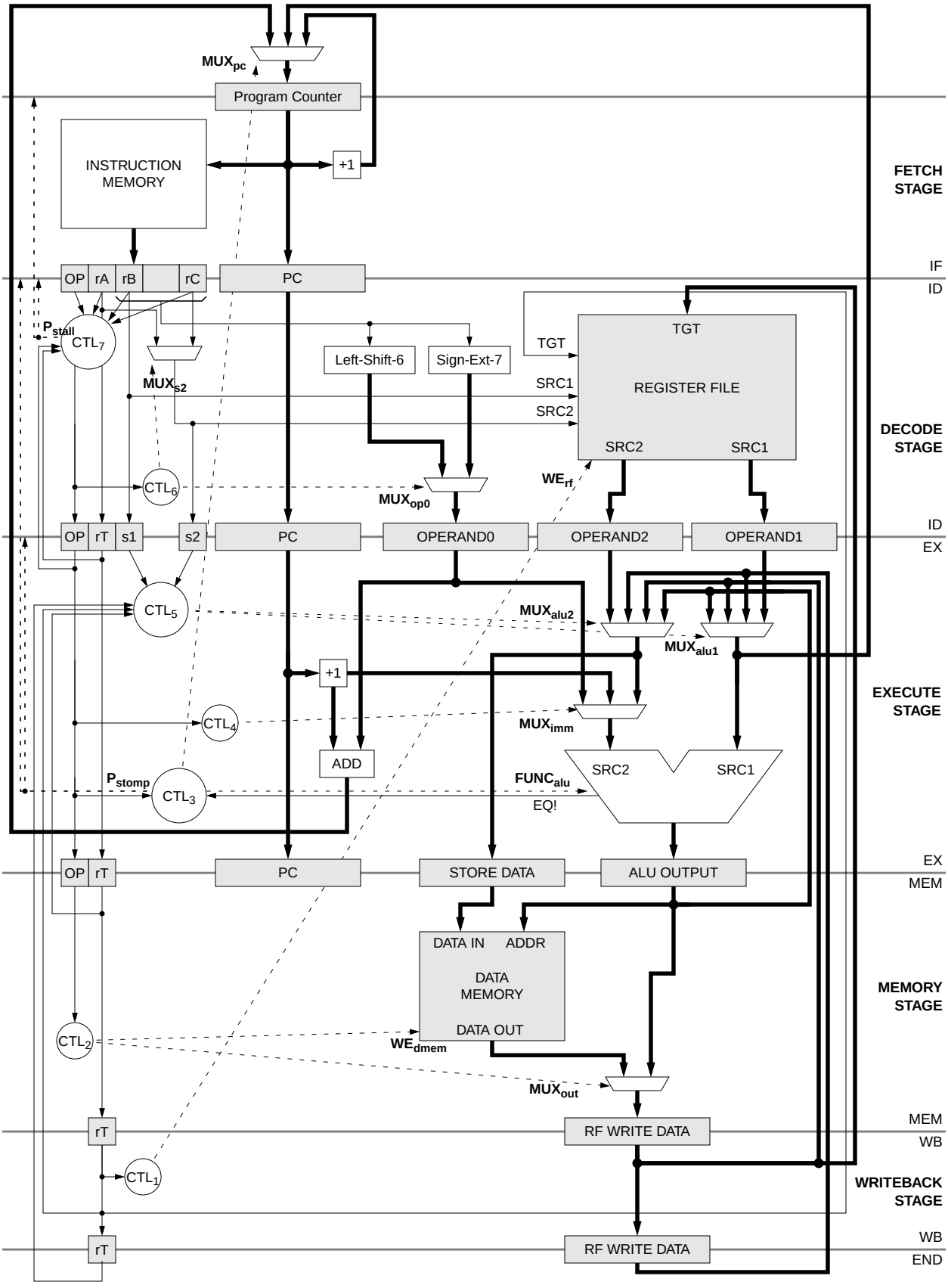


Fig. 1: RiSC-16 5-stage pipeline

forwarding is impossible, unless the register file has a pass-through design that connects data-in to data-out whenever reading and writing the same register. If the register file does not do such forwarding, then the data written to the register file is only available on the following cycle. Thus, there must be a path to forward data to the instruction in the decode stage at the same time as the instruction writing to the register file in the writeback stage. This is the function of the WB/END register and the forwarding stage it represents.

Pipeline Registers

Program Counter The address of the instruction currently being fetched.

IF/ID Register:

INSTR The instruction to execute, with opcode, rA, rB, rC, and immediate fields.

PC Contains the address of the instruction whose state is contained in this pipeline register. This is used by BEQ and JALR instructions and in handling pipeline interrupts.

ID/EX Register:

OP Contains the instruction opcode.

rT, s1, s2 Contains the instruction's 3-bit register specifiers *rA*, *rB*, and *rC—s1* is equal to *rB*. For SW and BEQ instructions, *s2* is taken from the instruction's *rA* field, otherwise it is taken from the instruction's *rC* field. The *rT* field contains the instruction's 3-bit target-register identifier, or the binary value 000 if the instruction has no target (i.e. SW and BEQ instructions).

PC Contains the address of the instruction whose state is contained in this pipeline register. This is used by BEQ and JALR instructions and in handling pipeline interrupts.

OPERAND0 Contains the instruction's immediate operand. If the instruction uses a shifted or sign-extended immediate value (ADDI, LUI, LW, SW, BEQ), that value is available immediately and is stored here.

OPERAND1 Contains the instruction's first register operand; this is the contents of the register *register-file*[*rB*].

OPERAND2 Contains the instruction's second register operand. For ADD and NAND instructions, it is the contents of *register-file*[*rC*]. For BEQ and SW instructions, it is the contents of *register-file*[*rA*].

EX/MEM Register:

OP Contains the instruction opcode.

rT Contains the instruction's 3-bit target-register identifier, or the binary value 000 if the instruction has no target (i.e. SW and BEQ instructions).

PC Contains the address of the instruction whose state is contained in this pipeline register. This is used by BEQ and JALR instructions and in handling pipeline interrupts.

STORE DATA Contains the data to store to DATA MEMORY. Note that if the instruction is not a SW, this information is not used.

ALU OUTPUT Contains the most recent output of the ALU.

MEM/WB Register:

rT Contains the instruction's 3-bit target-register identifier, or the binary value 000 if the instruction has no target (i.e. SW and BEQ instructions).

RF WRITE Contains the data that will be written to the register file on the following cycle (provided the **rT** register has a non-zero value).

WB/END Register:

rT Contains the instruction's 3-bit target-register identifier, or the binary value 000 if the instruction has no target (i.e. SW and BEQ instructions).

RF WRITE Contains the data that was written to the register file on the previous cycle (provided the **rT** register has a non-zero value).

Control Modules

These are the descriptions of the various CONTROL modules.

CTL_1 This module controls the write-enable line of the register file. If any data is to be written to the register file, it comes from the MEM/WB register. Thus, the control logic simply looks at the **rT** register: if that register is zero, write-enable (WE) is turned off. Otherwise, write-enable is turned on.

CTL_2 This module controls both the write-enable line of the data memory and the operation of MUX_{out}, which feeds the RF WRITE DATA register and therefore determines what will be written to the register file on the following cycle. Thus, the only input to the control module is the opcode of the instruction. The write-enable line of the data memory is only set if the opcode is SW; otherwise, writing is disabled. MUX_{out} only chooses the output of the data memory if the opcode is LW; otherwise, the mux chooses the value of the ALU OUTPUT register in EX/MEM.

CTL_3 This module controls the operation of the ALU, the operation of MUX_{pc}, and the STOMP logic. The module's inputs are the instruction opcode and the EQ! line returning from the ALU, which indicates if the two input operands are equal. The $FUNC_{alu}$ control line is determined solely from the instruction opcode. The value of MUX_{pc} is set by the opcode and the EQ! signal: if the instruction is a BEQ and the operands are equal (or if it is determined that the branch was mis-speculated, if more sophisticated branch prediction is implemented), MUX_{pc} chooses the value of the PC+1+OPERAND0 adder in the execute stage. When this happens, the contents of the IF/ID and ID/EX registers are overwritten with NOP instructions (this is a STOMP event). If the instruction is a JALR, MUX_{pc} chooses the output of the ALU SRC1 mux and also enables a STOMP event. For all other instructions and instances, MUX_{pc} chooses the output of the PC+1 register in IF/ID and no STOMP event occurs.

CTL_4 This module controls the operation of MUX_{imm}, the multiplexer that chooses the value to be sent to the ALU's SRC2 input. The control input is the instruction's

opcode. For all instructions that use immediate values, the value in OPERAND0 is chosen. JALR also uses the output of the PC+1 adder, as it takes the value PC+1 and places it into the register file. For all other instructions (ADD, NAND), the mux chooses OPERAND2 (or, more accurately, the output of the ALU2 mux).

CTL_5 This module handles data forwarding; it controls the operation of MUX_{alu1} and MUX_{alu2}, the two muxes responsible for forwarding data from pipeline registers further down the pipe. The control module's input is the register identifiers **s1** and **s2** of the instruction currently in the execute stage and the **rT** identifiers of the previous three instructions. The control module compares each of the current instruction's input register operands against the output of the previous three instructions. If it is determined that any of the previous three instructions write to any of the registers that the current instruction uses as operands, and if the register specifier in question is non-zero, the data is forwarded from the appropriate pipeline register, giving priority to instructions in higher stages (instructions nearer in time to the current instruction).

CTL_6 This module controls the operation of MUX_{op0}, the mux responsible for the contents of the **OPERAND0** field of the ID/EX register, and MUX_{s2}, the mux responsible for choosing between the rA and rC instruction fields for specifying the second register operand. It also simplifies the logic for CTL_5. The control module's input is the opcode of the instruction currently in the decode stage. MUX_{op0} chooses between the sign-extended immediate value (to be used for ADDI, LW, SW, BEQ, and JALR instructions) and the left-shifted immediate value (to be used for LUI instructions). MUX_{s2} chooses rC for ADD and NAND instructions; it chooses rA for all others. The control module simplifies the logic for CTL_5 by placing either rA or rC into the ID/EX register's **s2** field, so that, for BEQ and SW instructions, **s2** in ID/EX contains the appropriate bits (**s2** essentially latches the output of the MUX_{s2} multiplexer). This arrangement simplifies CTL_5 by eliminating the need to look at both rA and rC and choose, based on OP.

CTL_7 This module handles the load-use interlock, sets the register target **rT** in ID/EX. Its inputs are the opcode and register operand specifiers of the instruction currently in the decode stage (held in the IF/ID register) and the opcode and target register *rA* of the instruction in the execute stage. If the instruction currently in the execute stage (held in the ID/EX register) is a LW and targets any register that the instruction in decode uses as a source register, a STALL event is created. The control module's outputs are the **OP**, **rT**, and **s2** fields of the ID/EX register, and the $\mathbf{P_{stall}}$ signal, which directs the PC and IF/ID pipeline registers to not latch new values on the next cycle but to retain their values instead. On a pipeline stall, the instructions in the fetch and decode stages are held up, and the rest of the instructions in the pipeline are allowed to move ahead; to fill the created hole, a NOP instruction is placed in the ID/EX register. This amounts to putting an ADD opcode with target register r0 into the **OP** and **rT** fields of ID/EX. The module produces a value for the **rT** register in ID/EX as follows: if the instruction in IF/ID is a type that targets the register file (ADD, ADDI, NAND, LUI, LW, JALR), the value of rA is passed on to the **rT** register. For SW and BEQ instructions, the binary value 000 is passed, indicating that the instruction does not store a value in the register file (this works because r0 is a read-only target).

Control Signals

There are a number of control signals that change the direction and flow of data in the pipeline. These are the signals that the various CONTROL modules export:

$FUNC_{alu}$ This signal instructs the ALU to perform a given function.

MUX_{alu1} This 2-bit signal controls the mux connected to the SRC1 input to the ALU. The mux chooses between the OPERAND1 component of the ID/EX register and the outputs of the previous three instructions, held in the EX/MEM, MEM/WB, and WB/END pipeline registers.

MUX_{alu2} This 2-bit signal controls the mux connected to the SRC2 input to the ALU. The mux chooses between the OPERAND2 component of the ID/EX register and the outputs of the previous three instructions, held in the EX/MEM, MEM/WB, and WB/END pipeline registers.

MUX_{imm} This 2-bit signal controls the mux connected to the SRC2 ALU input. The mux chooses between the register output, the immediate value, and the PC-based value. The PC-based value is chosen for JALR instructions; the immediate value is chosen for ADDI, LUI, LW, and SW instructions, and the register operand is chosen for all others (ADD, NAND, BEQ).

MUX_{op0} This 1-bit signal controls the mux connected to the OPERAND0 component of the ID/EX register. The mux chooses between the sign-extended immediate value (to be used for ADDI, LW, SW, and JALR instructions) and the left-shifted immediate value (to be used for LUI instructions).

MUX_{out} This 1-bit signal controls the mux connected to the RF WRITE DATA component of the MEM/WB register, which holds the data to be written to the register file on the following cycle (provided the write-enable bit of the register file is set). The mux chooses between the output of the ALU and the output of the data memory (for LW instructions).

MUX_{pc} This 2-bit signal controls the mux connected the PC. The mux chooses between the output of the ALU's SRC1 multiplexer (to be used for JALR instructions), the PC+1+OPERAND0 adder in the execute stage (for instances of BEQ instructions that are taken, or branch mispredicts if speculative execution is implemented), and the output of the dedicated adder that produces the sum PC+1 every cycle.

MUX_{s2} This 1-bit signal controls the mux connected to the register file's SRC2 operand specifier, a 3-bit signal that determines which of the registers will be read out onto the 16-bit SRC2 data output port. The same signal is also latched in the ID/EX register's **s2** field. The mux chooses between the rA and rC fields of the instruction word: rC is chosen for ADD and NAND instructions.

P_{stall} The *pipeline stall* signal. This 1-bit signal indicates that the PC and IF/ID pipeline registers should not latch new data on the next clock edge but instead retain their current contents. The signal causes a pipeline stall event, during which the instructions in the execute and later stages are allowed to move forward one stage, but the topmost two instructions (in fetch and decode stages) are held back.

P_{stomp} The *pipeline stomp* signal. This 1-bit signal indicates that the IF/ID and ID/EX pipeline registers should not latch on the next clock edge but instead latch NOP instruc-

tions. The signal is used to implement a branch-taken event (or branch-mispredict event, if speculative execution is implemented), in which a branch (BEQ or JALR) instruction in the execute stage changes the direction of control flow.

WE_{rf} This 1-bit signal enables or disables the write port of the register file. If the signal is high, the register file can write a result. If it is low, writing is blocked. It is high for ADD, ADDI, NAND, LUI, LW, and JALR instructions.

WE_{dmem} This 1-bit signal enables or disables the write port of the data memory. If the signal is high, the data memory can write a result. If it is low, writing is blocked. It is high for SW instructions.

Not Included

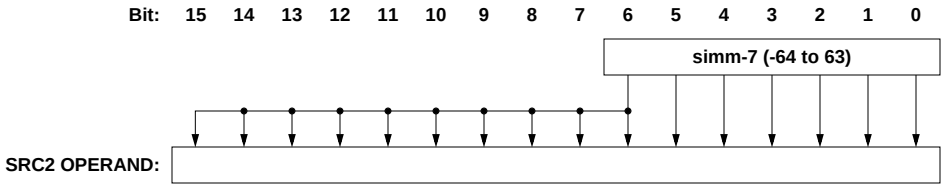
The design does not cover a number of issues involved in real-world implementations, including data caches, instruction caches, data- and instruction-cache misses, support for precise interrupts, or branch prediction more sophisticated than *predict-not-taken* (which is what the design implements). However, pipeline interrupts are described in the next section.

To implement caches, it is necessary to create paths to memory and allow for the possibility that the requested data or instruction word is not available (i.e. cache misses). For instruction cache misses, it is not necessary to stall the entire pipeline, but for data caches it is. This is because an instruction currently in the execute stage might be expecting a forwarding path to bring the most recent data from the MEM/WB or WB/END pipeline registers—if these instructions are allowed to continue down the pipeline while the instruction in the memory stage stalls for a data-cache miss, the data to be forwarded will be lost to the instruction in the execute stage. It will be in the register file, but the instruction in the execute stage (held in the ID/EX register) has already accessed the register file in the previous stage.

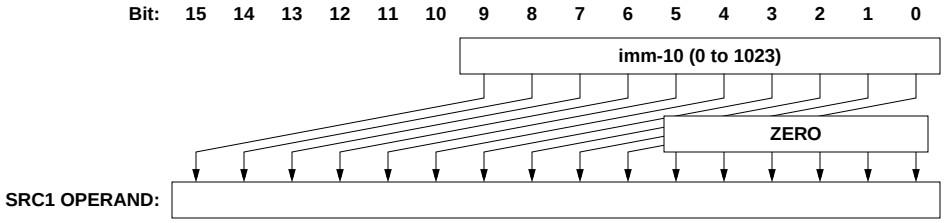
More sophisticated branch prediction is possible, requiring a few simple things. First, the IF/ID register needs a few extra bits holding branch-prediction state, including *BranchPredictorIndex, PredictedDirection, TakenDirection*, as well as the 16-bit *TakenBranchTarget*. It is important to retain this information as well as the PC through to the execute stage so that either a taken branch or a non-taken branch could be corrected if it is determined that either the choice of direction or the target itself was a misprediction. Note that, if the target is not known at prediction time, it might be that the direction predicted is *taken*, but because the branch cannot actually be taken without a predicted target, the prediction will be ignored. Also, note that the PC cannot be used during the execute phase as the branch-predictor index (as it is during fetch). Thus, to update the predictor's saturating counter, it is necessary to retain the branch-predictor table index or indices for this purpose.

Additional Logic

The Left-Shift-6 and Sign-Extend-7 logic components are identical to those described in the document *RiSC-16: Sequential Implementation*. Sign-Extend-7 extends the sign of the immediate value (as opposed to simply adding zeroes at the top) and in so doing produces a two's complement number. It is used for ADDI, LW, SW, and BEQ instructions. Its logic looks like this:



Left-Shift-6 is used for the LUI instruction and its logic looks like this:



3. Support for Precise Interrupts

Though the implementation of precise interrupts is not specified directly, the hooks are there, largely by the fact that the program counter is preserved down the entire pipeline. There are at least two possibilities for implementation:

1. Interrupts can be handled in much the same manner as STOMP logic: the program counter is redirected, and a subset of the instructions in the pipe are wiped out. The only difference would be that the subset of instructions to stomp would be determined dynamically. For instance, if the interrupt is for an invalid opcode (not applicable in this instruction-set, but good for an example), it can be determined in the decode stage—and instructions in the execute, memory, and writeback stages would be allowed to finish. If the interrupt is for something like divide-by-zero (also not applicable here), it would be determined in the execute stage, and only instructions in the memory and writeback stages would be allowed to commit. Instructions in the fetch and decode stages would be turned into NOPs. If the data-memory access uses an invalid address, only the instruction in the writeback stage would be allowed to commit. In addition, the implementation would need logic to resolve multiple simultaneous interrupts, giving priority to those further down the pipe. This also implies that the PC of the instruction must be carried down the pipe until the latest stage in which it is possible to cause an interrupt (e.g. up to and including the memory-access stage).

2. Interrupts can be handled in much the same way as a system with a reorder buffer: at the time of instruction commit. This means recognizing that an exceptional situation has occurred, holding that information with the instruction state (i.e. in the pipeline registers), and acting on it during the writeback stage—*only* in the writeback stage. Thus, in addition to extra fields in the pipeline register to hold interrupt type, the MEM/WB register also needs a copy of the program counter.

The primary differences between the two scenarios are simplicity and performance. The first scheme acts upon exceptional conditions as soon as they are detected, thereby saving a few cycles per interrupt, but it must also handle situations where an older instruction causes an interrupt *after* a newer instruction causes its own. In this case, the pipeline would be in the process of handling the newer instruction's exception when the older instruction's exception is detected. The pipeline must abort the interrupt-handler-in-progress and redirect control to handle the exception that was detected second but should be handled first (in program order). Thus, the first scheme requires a bit more logic.

For more details on the transfer of control in exceptional situations, see the document *RiSC-16 System Architecture.*