The RiSC-16 Instruction-Set Architecture

ENEE 646: Digital Computer Design, Fall 2002
Prof. Bruce Jacob

This paper describes a sequential 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]

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

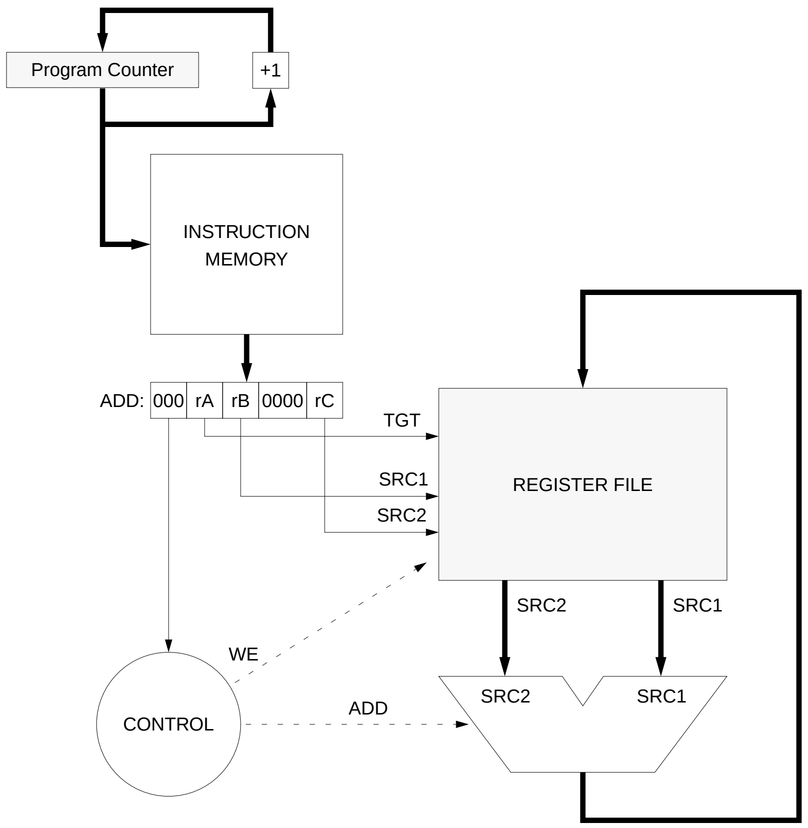
`jalr regA, regB`	PC <- R[regB], R[regA] <- PC + 1
PSEUDO-INSTRUCTIONS:	
`nop`	do nothing
`halt`	stop machine & print state
`lli regA, immed`	R[regA] <- R[regA] + (immed & 0x3f)
`movi regA, immed`	R[regA] <- immed
`.fill immed`	initialized data with value *immed*
`.space immed`	zero-filled data array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

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

2. RiSC-16 Instruction Control and Dataflow

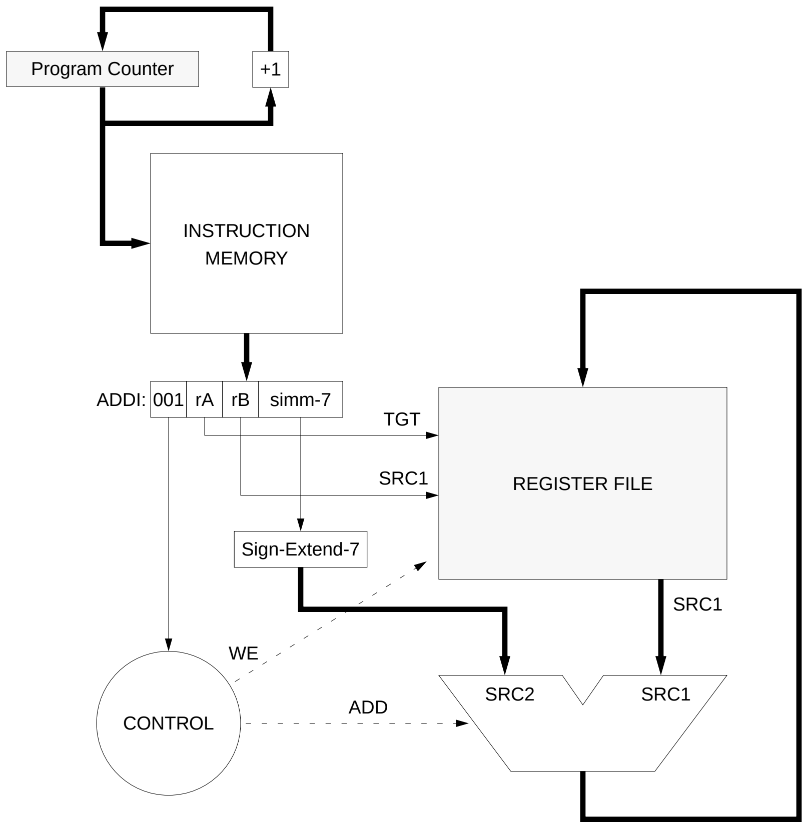
The following figures illustrate the flow of information for each of the instruction types for a simple sequential implementation—a scaled-down version of which has been done in discrete logic (i.e. 4-bit TTL parts) . A final figure will put all of the instructions together into one framework. Shaded boxes represent registers. Thick lines represent 16-bit buses.

ADD



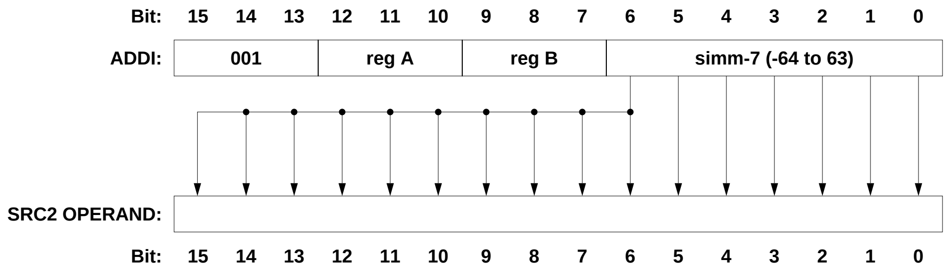
This figure illustrates the flow-of-control for the ADD instruction. All three ports of the register file are used, and the write-enable bit (WE) is set for the register file. The ALU control signal is a simple ADD function.

ADDI

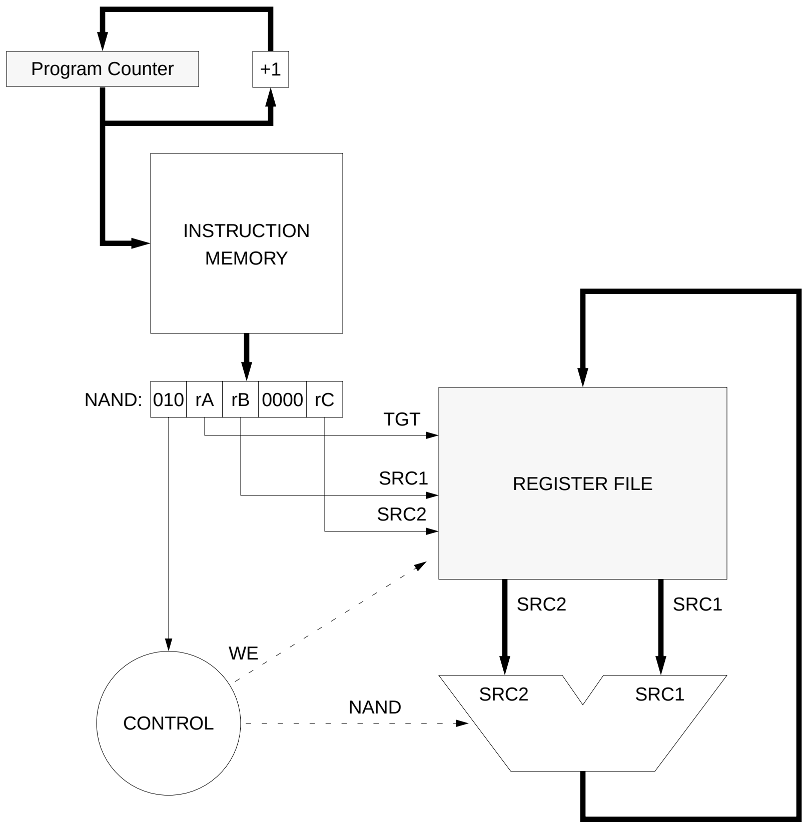


This figure illustrates the flow-of-control for the ADD IMMEDIATE instruction. Only two ports of the register file are used; the second source operand comes directly from the instruction word. The write-enable bit is set for the register file. The ALU control signal is a simple ADD function.

The Sign-Extend-7 logic 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. The logic looks like this:

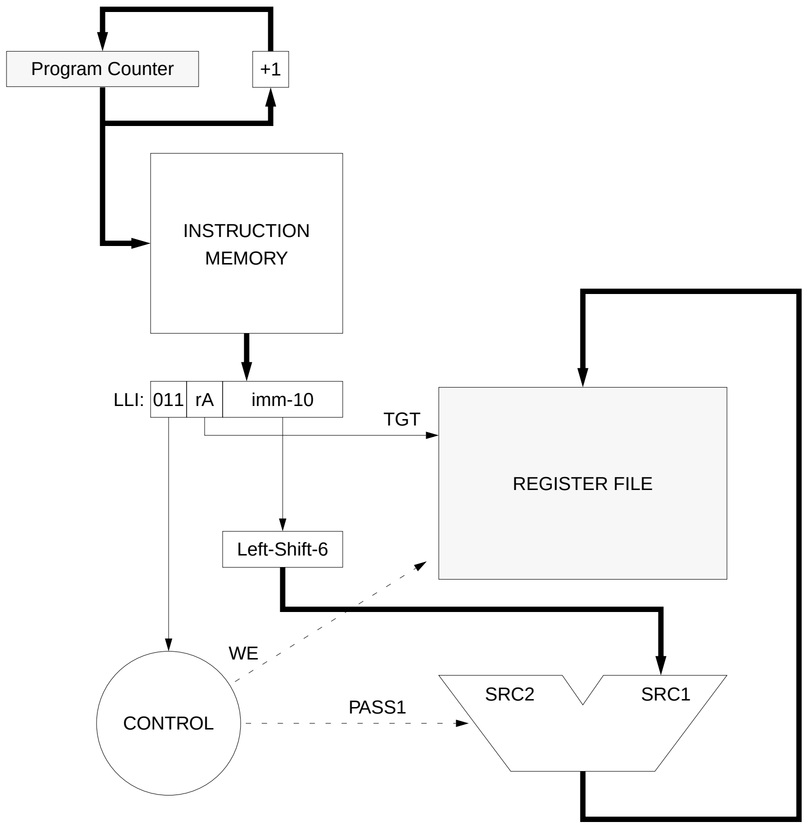

Bit: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
ADDI:
001
reg A
reg B
simm-7 (-64 to 63)
SRC2 OPERAND:
Bit: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

NAND



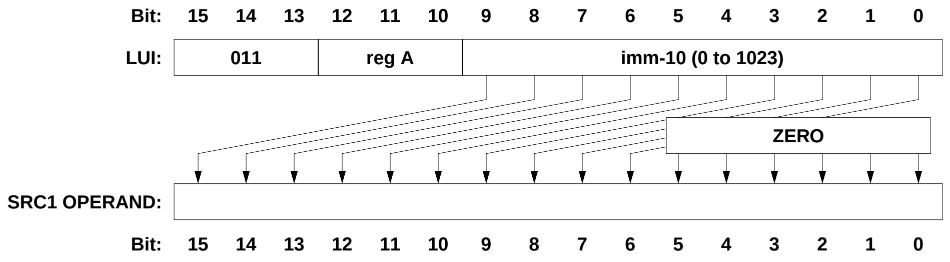
This figure illustrates the flow-of-control for the BITWISE NAND instruction. All three ports of the register file are used, and the write-enable bit is set for the register file. The ALU control signal is a BITWISE NAND function.

LUI



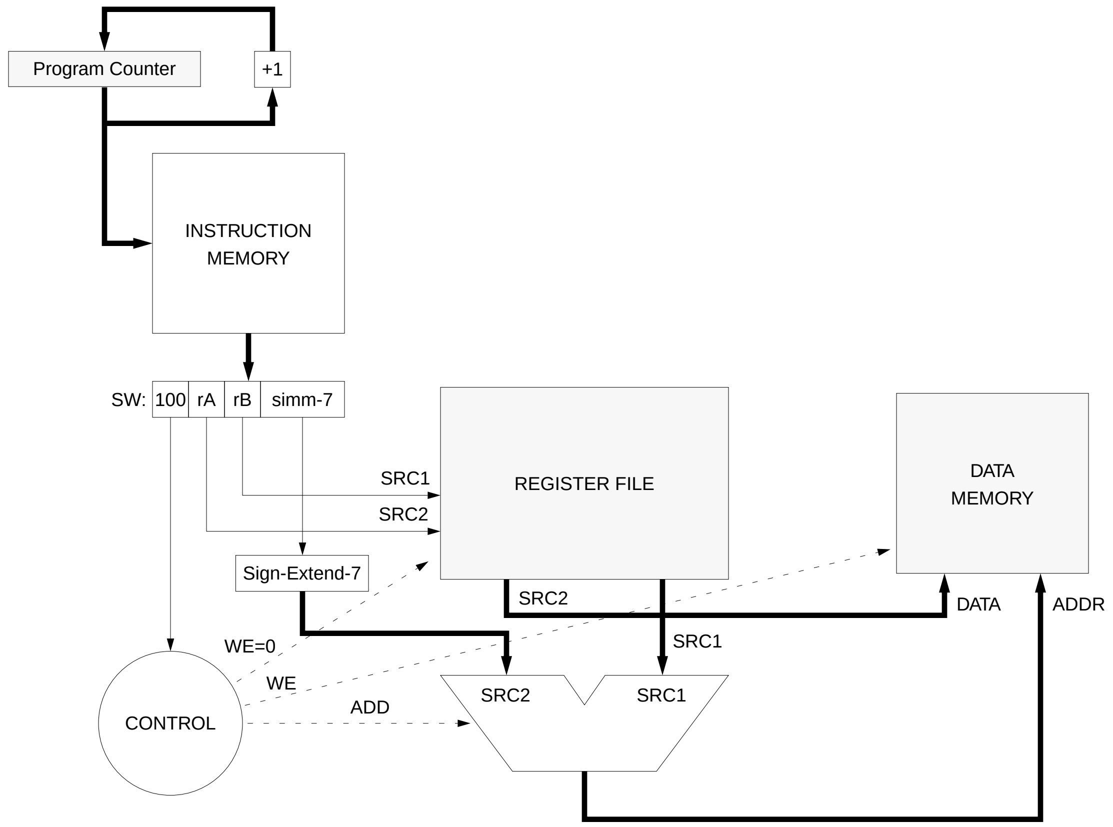
This figure illustrates the flow-of-control for the LOAD UPPER IMMEDIATE instruction. This instruction replaces the top ten bits of an instruction with ten bits found in the instruction word and zeroes out the bottom six bits. Only one port of the register file is used—the write port—as all of the bits destined for the register file are zero or come directly from the instruction word. The write-enable bit is set for the register file.

Compared to ADDI, the immediate-manipulation logic for LUI is slightly different; the logic for Left-Shift-6 looks like this:



The second argument to the ALU (SRC2) is ignored; this is accomplished by setting the ALU control to be PASS1, which sends through the first argument unchanged. This same ALU control will be used for the JALR instruction as well. One could alternatively zero-extend the immediate value and use a SHIFT input to the ALU while inputting the value ‘6’ at the other ALU-input. However, the illustrated mechanism seems slightly less complicated.

SW

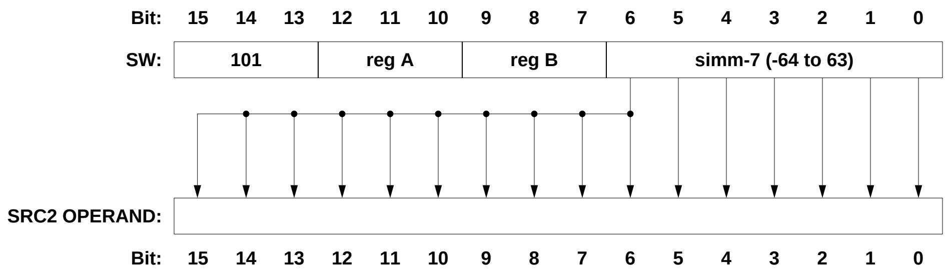


This figure illustrates the flow-of-control for the STORE DATA WORD instruction. Only two ports of the register file are used; the register file's write port (TGT) is not used. The SRC2 output port does not feed the ALU (as it normally does), but rather the DATA input of data memory. This is the value to be written to memory. The second ALU input is an immediate value that comes directly from the instruction word.

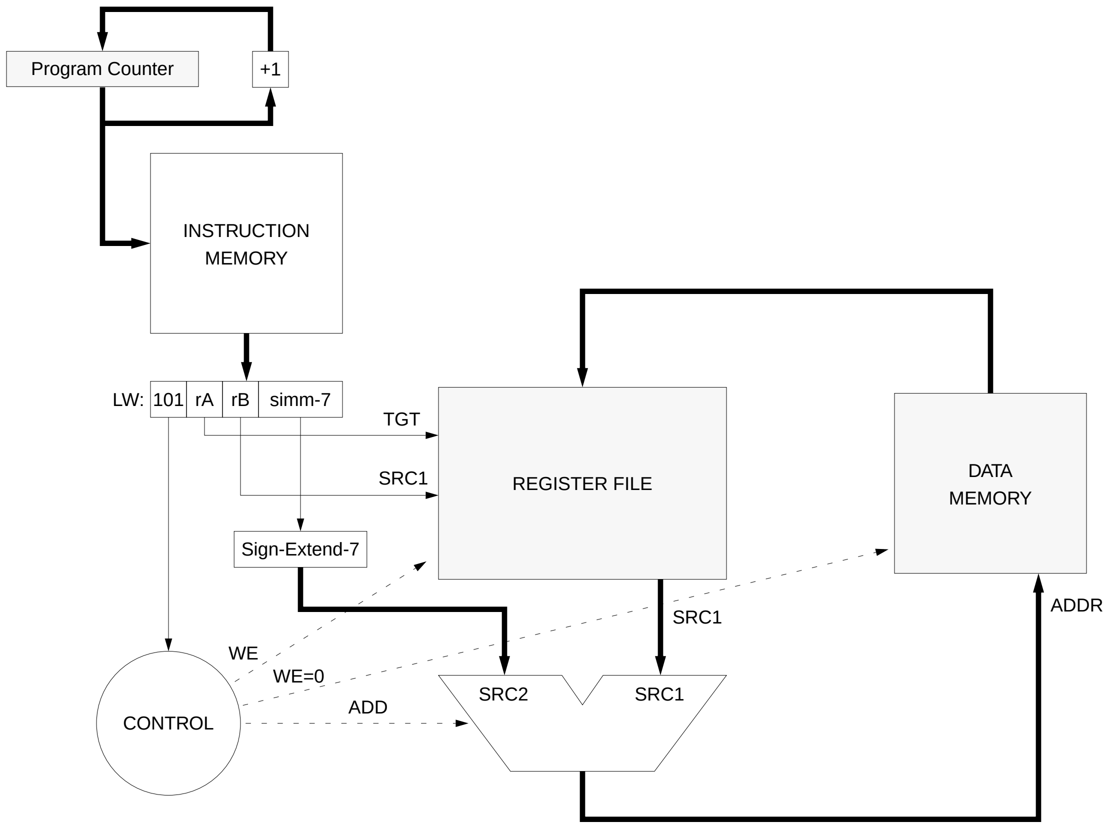
The write-enable bit is set for data memory but not for the register file. The ALU control signal is a simple ADD function. The result of the ALU ADD goes to the ADDR port of data memory, which responds by latching the data word at the specified address.

Note that the rA field of the instruction, which is normally tied to the register file's TGT specifier, is instead tied to the SRC2 specifier.

The Sign-Extend-7 logic 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. The logic looks like this:

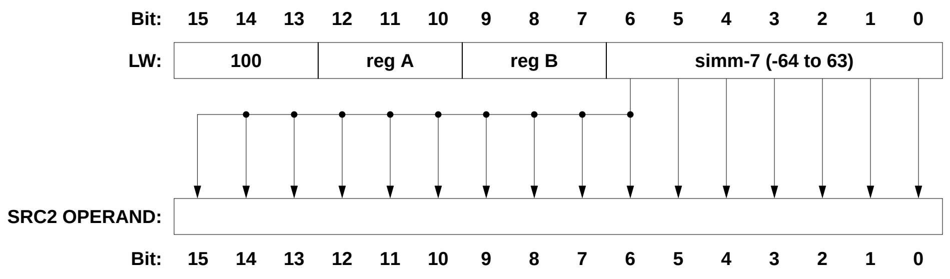

Bit: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
SW:
101
reg A
reg B
simm-7 (-64 to 63)
SRC2 OPERAND:
Bit: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

LW

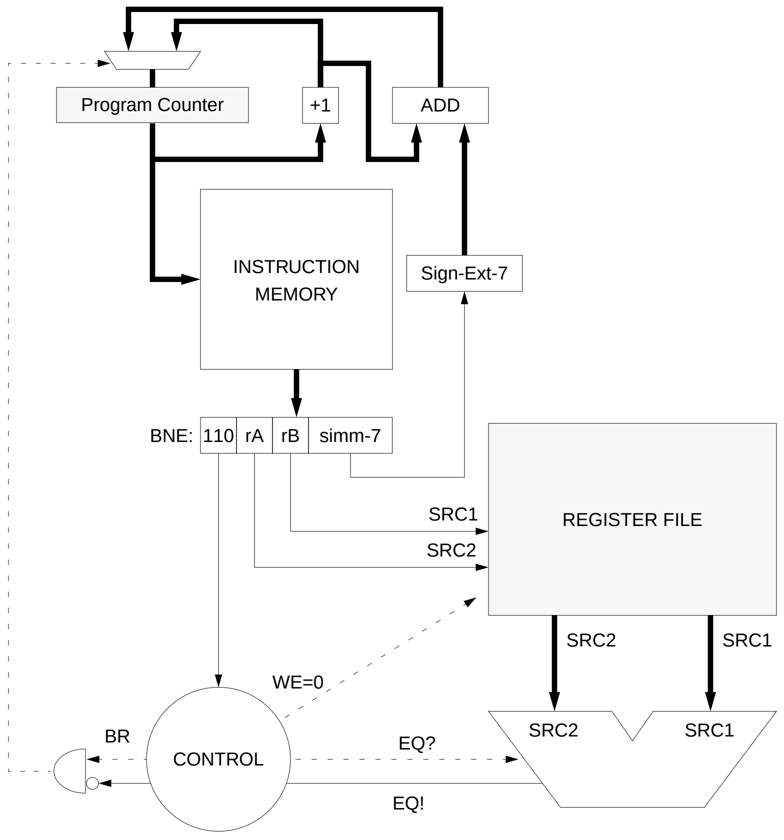


This figure illustrates the flow-of-control for the LOAD DATA WORD instruction. Only two ports of the register file are used; the second source operand comes directly from the instruction word. The write-enable bit is set for the register file but not for data memory. The ALU control signal is a simple ADD function. The result of the ALU ADD does not go to the register file but rather to the ADDR port of data memory, which responds by reading out the corresponding data word. This value is stored in the register file.

As in SW, the Sign-Extend-7 logic 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.


Bit: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
LW:
100
reg A
reg B
simm-7 (-64 to 63)
SRC2 OPERAND:
Bit: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

BNE



This figure illustrates the flow-of-control for the BRANCH-IF-NOT-EQUAL instruction. Like STORE DATA WORD instruction, it does not write a result to the register file. Two values are read from the register file and compared. The ALU control signal is a request for a test of equality (labeled EQ? in the diagram). If the ALU does not support an equality test, a SUBTRACT signal can be used, as most ALUs do have a 1-bit output signal that indicates a zero-value result: the two combined produce the same results as an EQ signal.

The result of the comparison determines which of two values are to be placed into the program counter. If the two values in the register file are **equal**, the value to be latched in the program counter is the sum

```
PC + 1
```

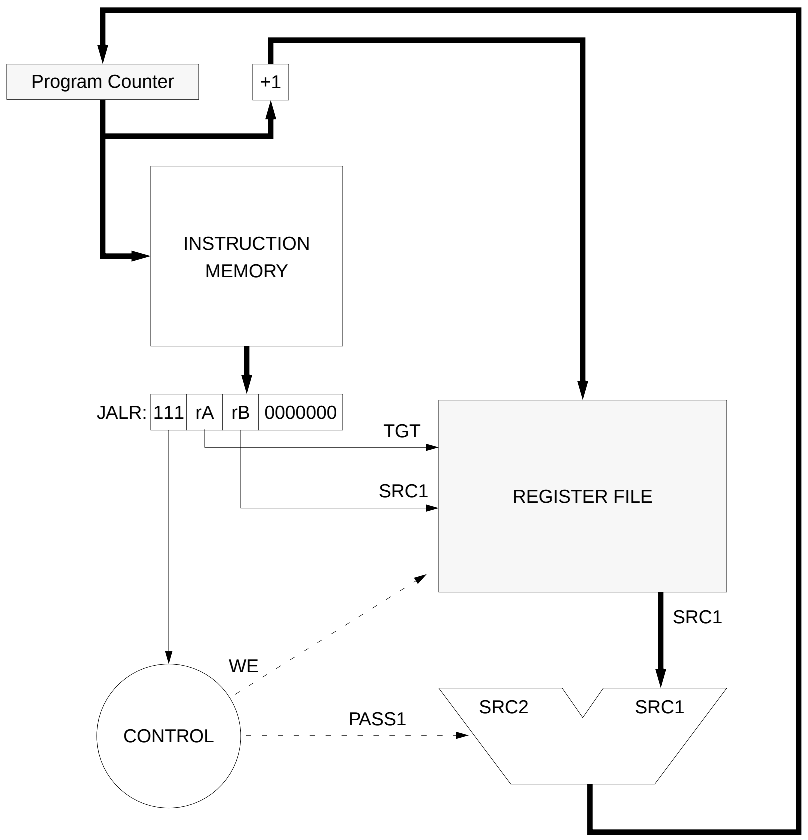
(which is the normal program counter update value). If the two values read from the register file are **NOT equal**, the value to be latched in the program counter is the sum

```
PC + 1 + (sign-extended immediate value)
```

(note that the sign extension is just like in LW and SW). We have shown a small portion of the CONTROL module: that portion controlling the PC mux. The choice between PC+1 and PC+1+IMM is represented by an AND of the negated EQ! output of the ALU and a signal that represents CONDITIONAL BRANCH (i.e. the 3-bit opcode is BNE:110). Thus, if the opcode is a **conditional branch** and the two values are **not equal**, choose PC+1+IMM, otherwise choose PC+1.

Note that the rA field of the instruction, which is normally tied to the register file's TGT specifier, is instead tied to the SRC2 specifier.

JALR



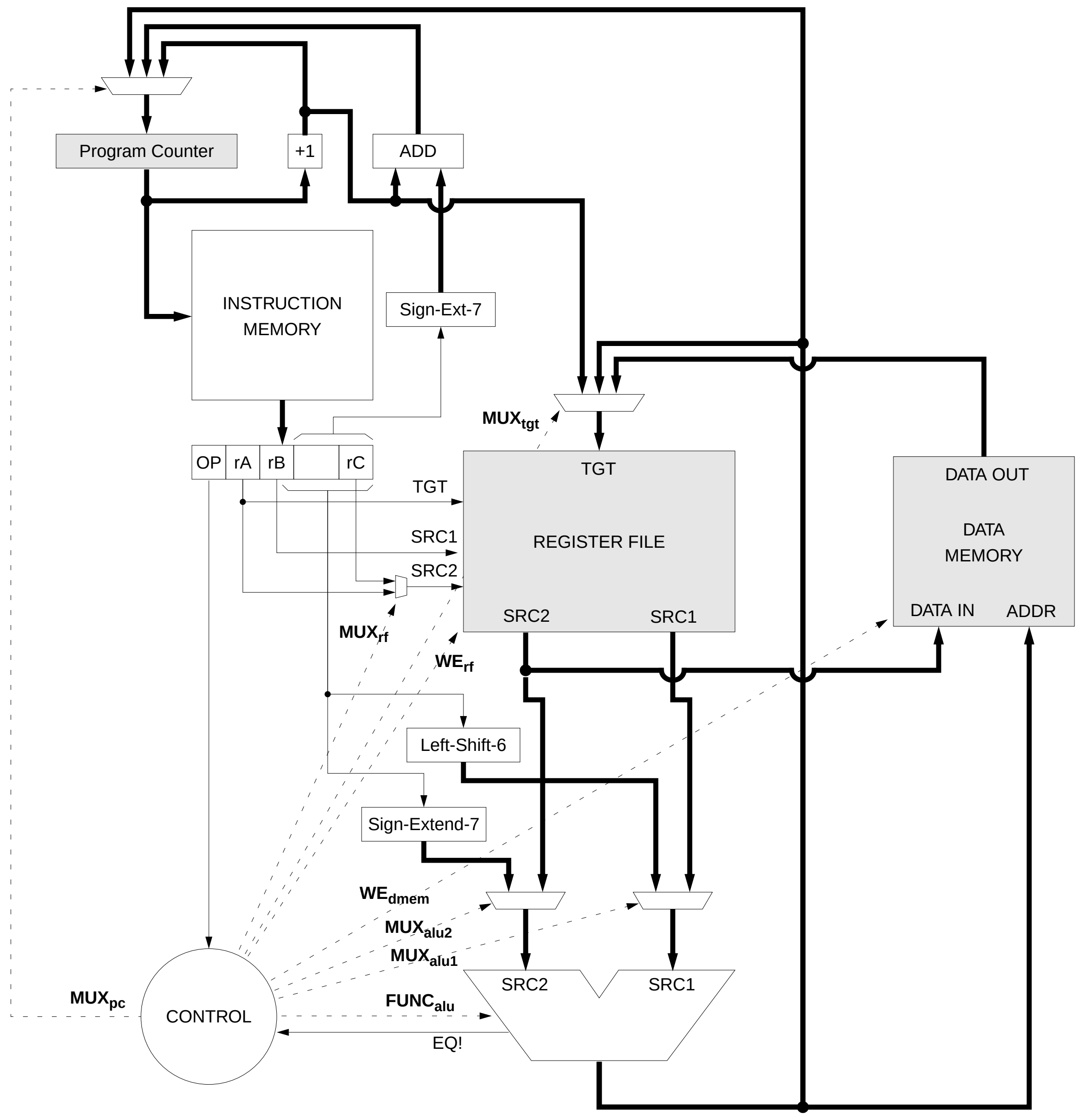
This figure illustrates the flow-of-control for the JUMP-AND-LINK-THROUGH-REGISTER instruction. This function uses two of the register file's three ports. A value is read from the register file and placed directly into the program counter, and the sum

```
    PC + 1
```

(which is normally latched into the program counter) is written to a specified register in the register file.

The ALU signal is a PASS signal for the SRC1 operand—the ALU is instructed to perform no function, only pass the SRC1 operand directly through. This value is written into the program counter.

Putting it All Together



This diagram combines all of the previous data and control paths into one complete framework. To accommodate all control and data paths, there are a number of muxes that choose the direction of data flow. These muxes are operated by the CONTROL module, which depends on only two inputs:

1. the 3-bit OPCODE of the instruction; and
2. the 1-bit EQ! signal from the ALU (which, for the BNE instruction, indicates that the two operands are equal—for all other instructions this signal is ignored).

The CONTROL module is essentially a decoder. It takes these input signals in and sets a number of outgoing signals that control the ALU, a host of multiplexers, and the write functions of the register file and data memory. At the beginning of every cycle, a new program counter value is latched, which causes a new instruction word to be read out and a new opcode to be sent to the

CONTROL module. The CONTROL module simply sets the control lines as appropriate for the instruction, and all of the control and data paths are set up: after delays through the register file, the ALU, the data memory, and the corresponding muxes, all of the signals stabilize. At this point, the newly produced values are latched (in the register file, the data memory and/or the program counter), and the new program counter causes a new instruction to be read from the instruction memory.

These are the signals that the CONTROL module exports:

$\mathbf{FUNC_{alu}}$	This signal instructs the ALU to perform a given function.
$\mathbf{MUX_{alu1}}$	This 1-bit signal controls the mux connected to the SRC1 input of the ALU. The mux chooses between the SRC1 output of the register file and the left-shifted immediate value (to be used for LUI instructions).
$\mathbf{MUX_{alu2}}$	This 1-bit signal controls the mux connected to the SRC2 input of the ALU. The mux chooses between the SRC2 output of the register file and sign-extended immediate value (to be used for ADDI, LW, and SW instructions).
$\mathbf{MUX_{pc}}$	This 2-bit signal controls the mux connected the program counter. The mux chooses between the output of the ALU, the output of the +1 adder that produces the sum PC+1 on every cycle
$\mathbf{MUX_{rf}}$	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 mux chooses between the rA and rC fields of the instruction word.
$\mathbf{MUX_{tgt}}$	This 2-bit signal controls the mux connected to the register file's 16-bit TGT data input port, which carries the data to be written to the register file (provided the write-enable bit of the register file is set). The mux chooses between the output of the ALU, the output of the data memory, and the output of the +1 adder tied to the program counter.
$\mathbf{WE_{rf}}$	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.
$\mathbf{WE_{dmem}}$	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.