2 ISA

The RISC-E architecture is based on a RISC-type MIPS instruction set, with some extended instructions to aid in the use of the peripherals attached to the processor.

The RISC-E architecture includes 28 general purpose registers (registers `R1`-`R28`). Register `R30` is used as a stack pointer, but if no stack operations are performed, it too could be considered a general purpose register. Register `R31` is used by `jal` (jump and link) instructions to store return IP values, and is by convention used for `jr` (jump register or jump return) instructions, though it, too, is available for general purpose use.

Use of register `R29` is limited to the following special purposes:

1) Writes to `R29` will write characters (ASCII, lower eight bits only) to the VGA controller as character output, if there is sufficient space in the VGA pixel buffer. Use of the `brvid` instruction allows a programmer to poll availability of this buffer.
2) Reads from `R29` will read characters from the keyboard input controller, if a new key has been depressed. If no key has been depressed, an unspecified value will be read from register `R29` (see `brchar` instruction).

The following registers represent the primary operands for the instruction set following:

Key
`ddddd` – 5 bit destination register
`aaaaa` – 5 bit source register A
`bbbbb` – 5 bit source register B
`ooooo` – variable length offset (dependent on instruction type)
`AAAAA` – 18 bit address
`xxxxx` – don't care field

2.1 Arithmetic instructions

Arithmetic instructions perform basic mathematic operations on registered operands and store the result into a register. Opcodes for arithmetic instructions begin with 0h.

`add`

Opcode 31 : 28	Mode 27 : 24	Destination 23 : 19	SrcA 18 : 14	SrcB 13 : 9	8 : 0
0000	0000	ddddd	aaaaa	bbbbb	xxxxxxxxx

ddddd ← aaaaa + bbbbb

```
add $d $a $b
```

Adds the values in $a and $b together and stores in $d. Works on two's compliment values and does not consider overflow.

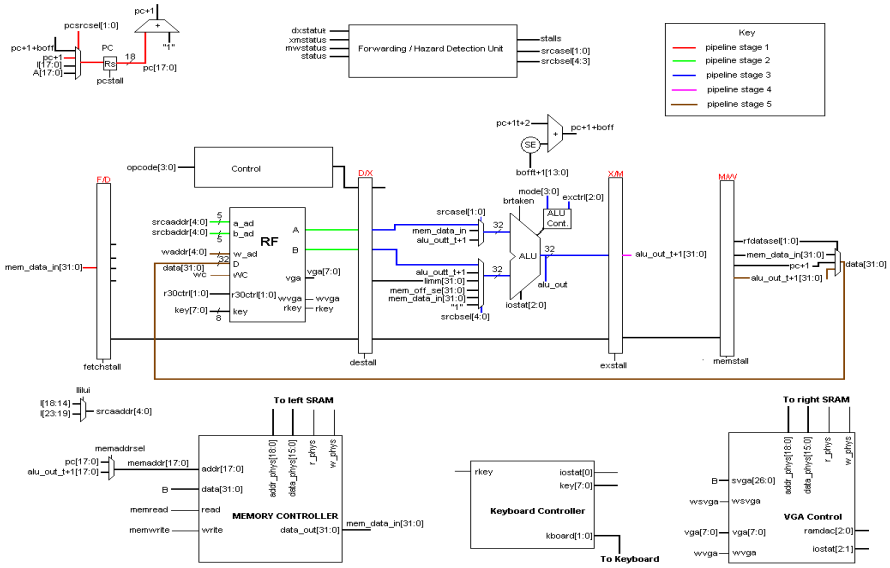
`sub`

Opcode 31 : 28	Mode 27 : 24	Destination 23 : 19	SrcA 18 : 14	SrcB 13 : 9	8 : 0
0000	0001	ddddd	aaaaa	bbbbb	xxxxxxxxx

ddddd ← aaaaa - bbbbb

```
sub $d $a $b
```

Subtracts the value in $b from the value in $a and stores the result in register $d. Works on two's compliment values and does not consider overflow.



Data flow for `add` and `sub` instructions.

`inc`

Opcode 31 : 28	Mode 27 : 24	Destination 23 : 19	SrcA 18 : 14	SrcB 13 : 9	8 : 0
0000	0010	ddddd	ddddd	xxxxx	xxxxxxxxx

ddddd ← ddddd + 1
`inc $d`

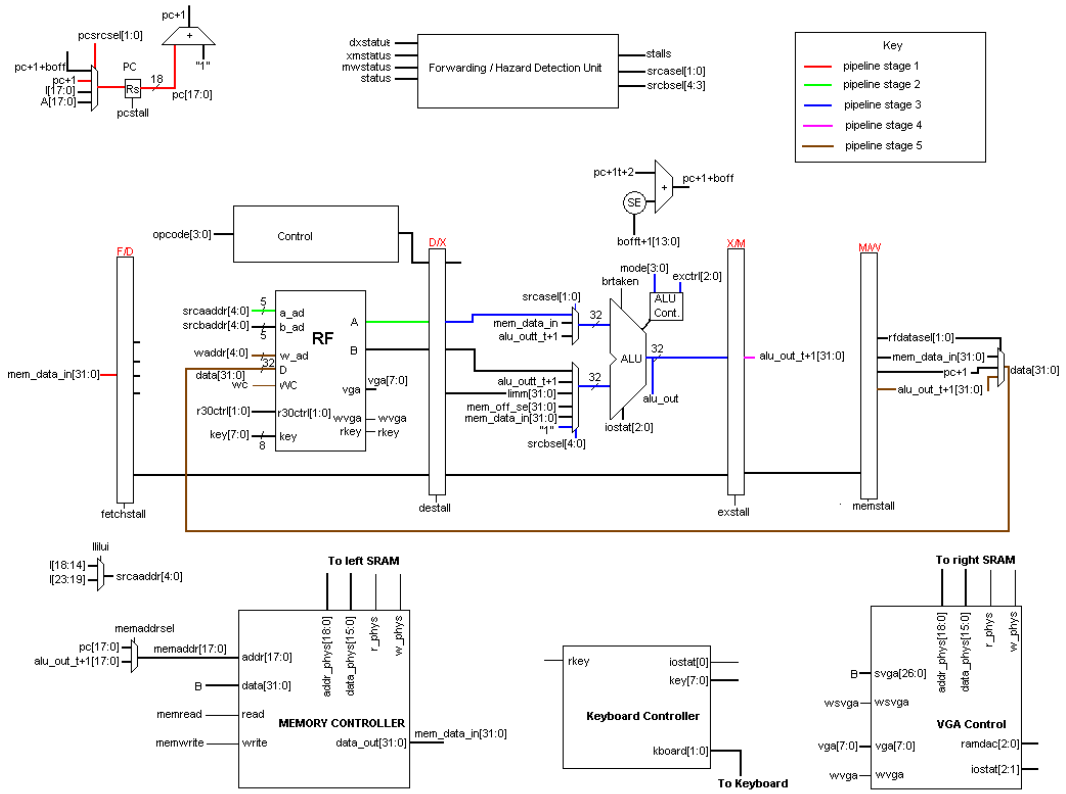
Increments the value in register $d by one and stores the result in register $d.

`dec`

Opcode 31 : 28	Mode 27 : 24	Destination 23 : 19	SrcA 18 : 14	SrcB 13 : 9	8 : 0
0000	0011	ddddd	ddddd	xxxxx	xxxxxxxxx

ddddd ← ddddd – 1
`dec $d`

Decrements the value in register $d by one and stores the result in $d.



Data flow for `inc` and `dec`.

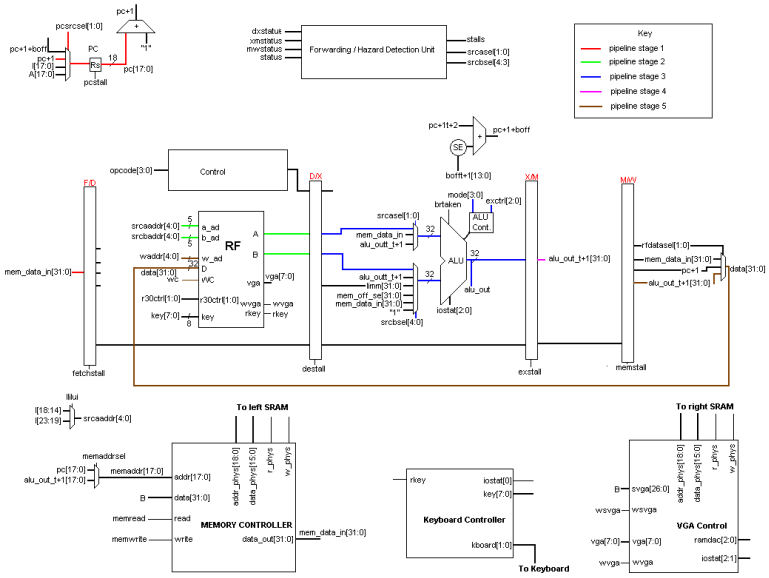
mult

Opcode 31 : 28	Mode 27 : 24	Destination 23 : 19	SrcA 18 : 14	SrcB 13 : 9	8 : 0
0011	0000	ddddd	aaaaa	bbbbb	xxxxxxxxx

ddddd ← aaaaa * bbbbb

```
mult $d $a $b
```

Multiplies the lower 16 bits of registers $a and $b and stores the 32 bit result in register $d. Works on two's compliment values.



Data flow for `mult`.

2.2 Logical Instructions

Logical instructions perform basic logical operations on register operands and store their results in a destination register.

`and`

Opcode 31 : 28	Mode 27 : 24	Destination 23 : 19	SrcA 18 : 14	SrcB 13 : 9	8 : 0
0001	0000	ddddd	aaaaa	bbbbb	xxxxxxxxx

ddddd ← aaaaa ● bbbbb

```
and $d $a $b
```

Performs a bitwise logical AND operation on registers $a and $b and stores the result to register $d. Stores 1 to a given bit if and only if both corresponding bits in $a and $b are 1, 0 otherwise.

`or`

Opcode 31 : 28	Mode 27 : 24	Destination 23 : 19	SrcA 18 : 14	SrcB 13 : 9	8 : 0
0001	0001	ddddd	aaaaa	bbbbb	xxxxxxxxx

ddddd ← aaaaa | bbbbb
`or $d $a $b`

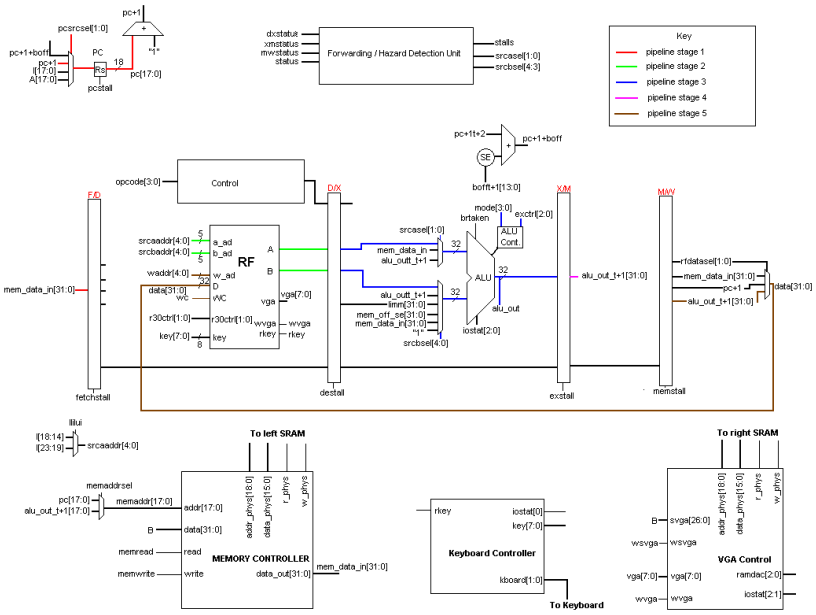
Performs a bitwise logical or operation on registers $a and $b and stores the result to register $d. Stores 0 to a given bit if and only if both corresponding bits in $a and $b are 0, 1 otherwise.

`xor`

Opcode 31 : 28	Mode 27 : 24	Destination 23 : 19	SrcA 18 : 14	SrcB 13 : 9	8 : 0
0001	0010	ddddd	aaaaa	bbbbb	xxxxxxxxx

ddddd ← aaaaa ^ bbbbb
`xor $d $a $b`

Performs a bitwise logical xor operation on registers $a and $b and stores the result to register $d. Stores 1 to a given bit if and only if both corresponding bits in $a and $b are different, 0 if they are the same.



Data flow for `and`, `or`, and `xor`.

`not`

Opcode 31 : 28	Mode 27 : 24	Destination 23 : 19	SrcA 18 : 14	SrcB 13 : 9	8 : 0
0001	0011	ddddd	aaaaa	xxxxx	xxxxxxxxx

ddddd ← ~aaaaa
`not $d $a`

Performs a bitwise logical not operation on register $a and stores the result to register $d. Stores a 1 if the corresponding bit in $a is a 0, and a 0 if the corresponding bit is a 1.

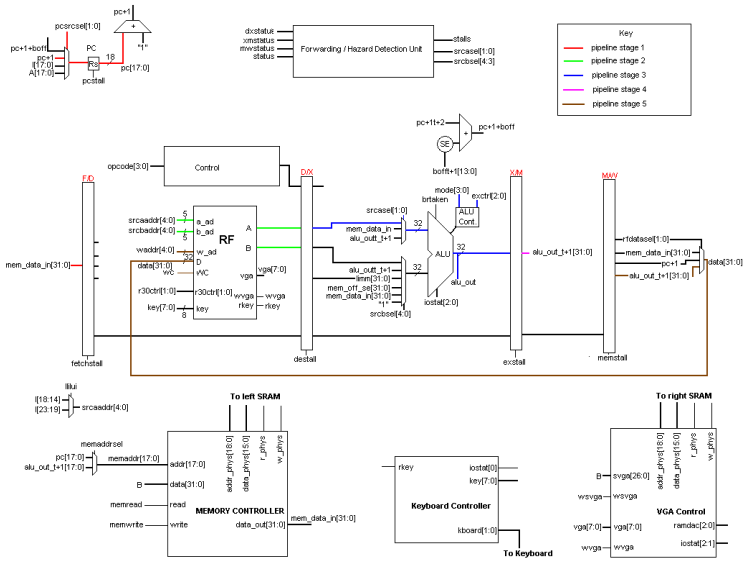


Figure : Picture of data flow for `not`.

2.3 Shift Instructions

Shift instructions move the source register by a value specified in the second source register.

`sra`

Opcode 31 : 28	Mode 27 : 24	Destination 23 : 19	SrcA 18 : 14	SrcB 13 : 9	8 : 0
0010	0000	ddddd	aaaaa	bbbbb	xxxxxxxxx

ddddd ← { (a[31])^(B%32) , a[31:(B%32)] }
`sra $d $a $b`

Shifts register $a to the right by the value in the lower 5 bits of register $b and stores the result in register $d. `sra` sign extends the value of $a.

`srl`

Opcode 31 : 28	Mode 27 : 24	Destination 23 : 19	SrcA 18 : 14	SrcB 13 : 9	8 : 0
0010	0001	ddddd	aaaaa	bbbbb	xxxxxxxxx

ddddd ← { 0^(B%32) , a[31:(B%32)] }
`srl $d $a $b`

Shifts register $a to the right by the value in the lower 5 bits of register $b and stores the result in register $d. `srl` does not sign extend the value of $a.

`sl`

Opcode 31 : 28	Mode 27 : 24	Destination 23 : 19	SrcA 18 : 14	SrcB 13 : 9	8 : 0
0010	0010	ddddd	aaaaa	bbbbb	xxxxxxxxx

ddddd ← { a[31-(B%32):0], 0^(B%32) }
`sl $d $a $b`

Shifts register $a to the left by the value in the lower 5 bits of register $b and stores the result in register $d.

`rol`

Opcode 31 : 28	Mode 27 : 24	Destination 23 : 19	SrcA 18 : 14	SrcB 13 : 9	8 : 0
0010	0011	ddddd	aaaaa	bbbbb	xxxxxxxxx

ddddd ← { a[(B%32)+1,0] , a[31:(B%32)] }
`rol $d $a $b`

Shifts register $a to the left by the value in the lower five bits of register $b and fills the lower bits of register $a with the bits of register $a that were shifted out and stores the result in register $d.

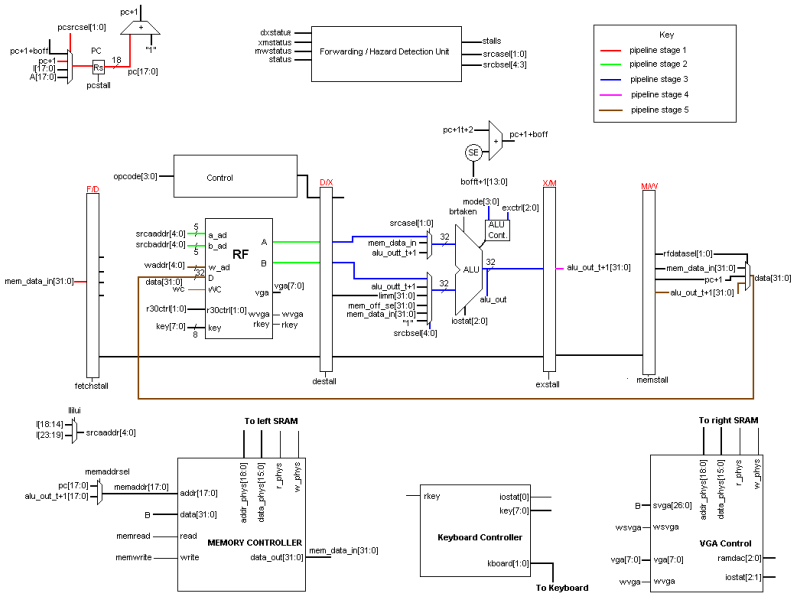
`ror`

Opcode 31 : 28	Mode 27 : 24	Destination 23 : 19	SrcA 18 : 14	SrcB 13 : 9	8 : 0
0010	0100	ddddd	aaaaa	bbbbb	xxxxxxxxx

ddddd ← { a[31:(B%32)], a[(B%32)+1,0] }

`ror $d $a $b`

Shifts register $a to the right by the value in the lower five bits of register $b and fills the upper bits of register $a with the bits of register $a that were shifted out and stores the result in register $d.



Data flow for `sra`, `srl`, `sl`, `rol`, and `ror`.

2.4 No-Operation

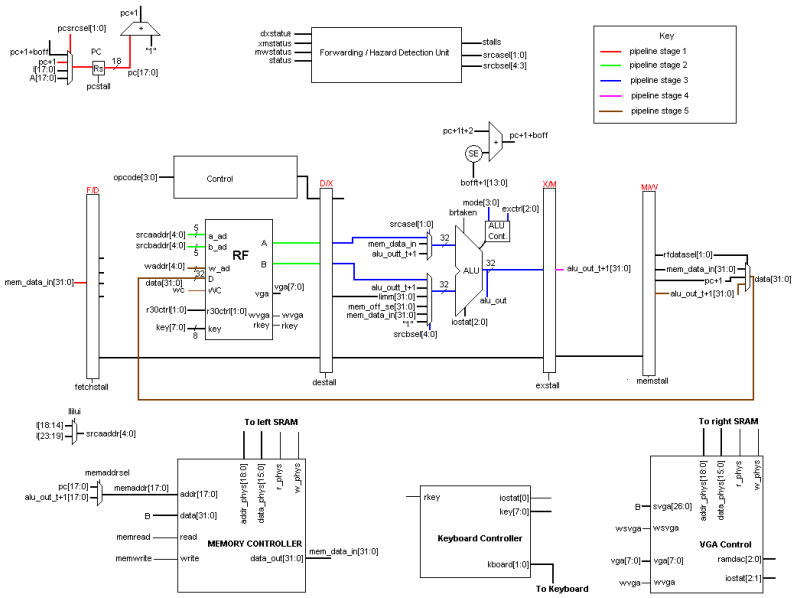
No operation performs the addition of register $0 to register $0 and stores the result in register $0.

`nop`

Opcode 31 : 28	Mode 27 : 24	Destination 23 : 19	SrcA 18 : 14	SrcB 13 : 9	8 : 0
0100	0000	00000	00000	00000	xxxxxxxxx

`nop`

No operation performs the addition of register $0 to register $0 and stores the result in register $0.



Data flow for `nop`.

2.5 Immediate Instructions

Immediate instructions load values specified in offset fields into the register operand.

`lli`

Opcode 31 : 28	Mode 27 : 24	Destination 23 : 19	18 : 16	Imm 15 : 8	Imm 7 : 0
0101	0000	ddddd	xxx	iiiiiiii	iiiiiiii

ddddd ← { ddddd[31:16] , i16 }

```
lli $d iiiih
```

Load lower immediate concatenates the upper 16 bits of register $d with the 16 bits of the immediate offset and stores the result in register $d.

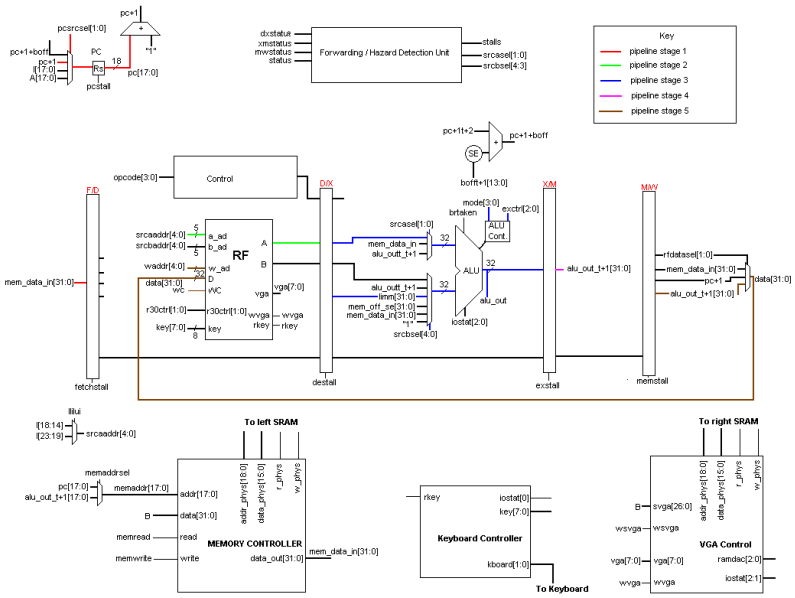
`lui`

Opcode 31 : 28	Mode 27 : 24	Destination 23 : 19	18 : 16	Imm 15 : 8	Imm 7 : 0
0101	0001	ddddd	xxx	iiiiiiii	iiiiiiii

ddddd ← { i16, ddddd[15:0] }

```
lui $d iiiih
```

Load upper immediate concatenates the lower 16 bits of register $d with the 16 bits of the immediate offset and stores the result in register $d.



Data flow for `lli` and `lui`.

2.6 Memory Instructions

Memory instructions manipulate memory locations by writing to or reading from them.

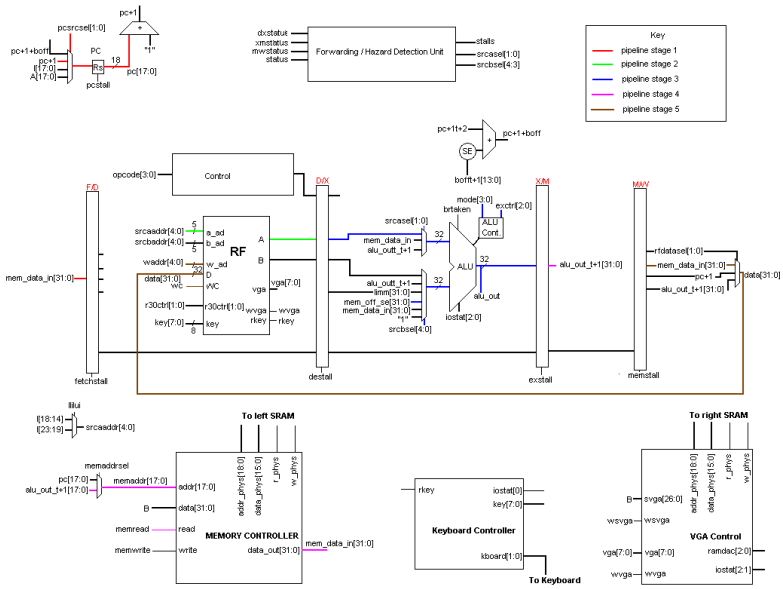
`lw`

Opcode 31 : 28	Mode 27 : 24	Destination 23 : 19	SrcA 18 : 14	 13 : 9	Offset 8 : 0
1000	0000	ddddd	aaaaa	xxxxx	ooooooooo

ddddd ← MEM[aaaaa + ooooooooo]

```
lw $d $a o
```

Load word reads the memory location specified by the value of register $a plus the offset and stores the result in register $d. Load word creates a unique hazard in the processor, as the MIU runs only fast enough to perform one memory access per clock cycle. Therefore, when a load word is encountered, the pipeline stalls for a cycle to perform the memory read.



Data flow for `lw`.

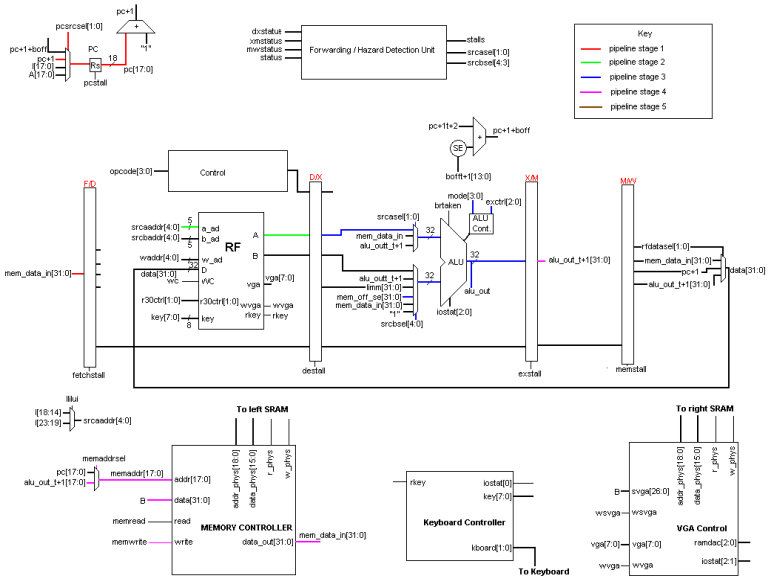
`sw`

Opcode 31 : 28	Mode 27 : 24	 23 : 19	SrcA 18 : 14	SrcB 13 : 9	Offset 8 : 0
1001	0000	xxxxx	aaaaa	bbbbb	ooooooooo

MEM[aaaaa + ooooooooo] ← bbbbb

```
sw $b $a o
```

Store words places the value of register $b in the memory location specified by the value of register $a plus the offset. Store word also creates a hazard in the processor, similar to that of load word. When a store word is encountered, the pipeline stalls for a cycle to perform the memory write.



Data flow for `sw`.

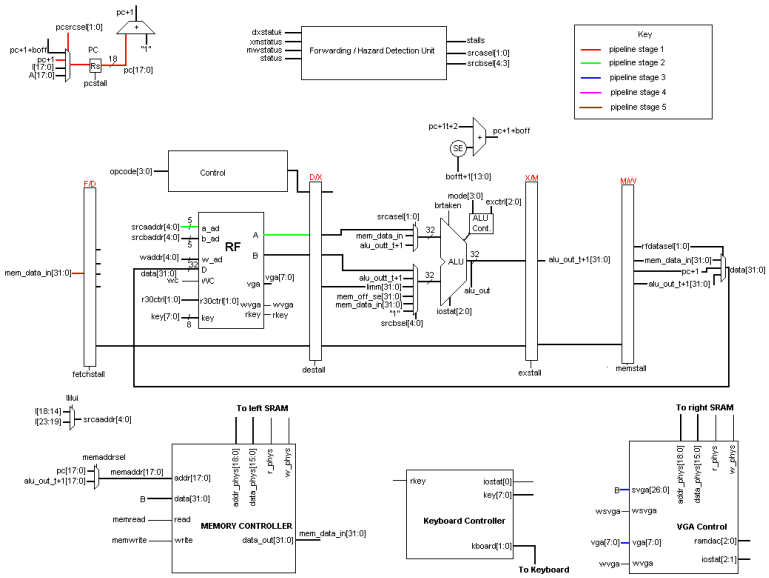
`svga`

Opcode 31 : 28	Mode 27 : 24	23 : 19	18 : 14	SrcB 13 : 9	8 : 0
1001	0001	xxxxx	xxxxx	bbbbb	xxxxxxxxx

VGAMEMBUFFNEXT ← b

```
svga $b
```

The Store to VGA instruction loads the value in register $b into the VGA buffer queue assuming that the buffer isn't full (see `brpix` instruction). The lower 8 bits of register $b are the color, 3 red bits, 2 green bits, and 3 blue bits. The next 20 bits are used to specify the location of the bit to be manipulated (See **VGA Unit**).



Data flow for `svga`.

2.7 Stack Instructions

Stack instructions load and store values into the address specified by register $30 and increment or decrement register $30 as needed.

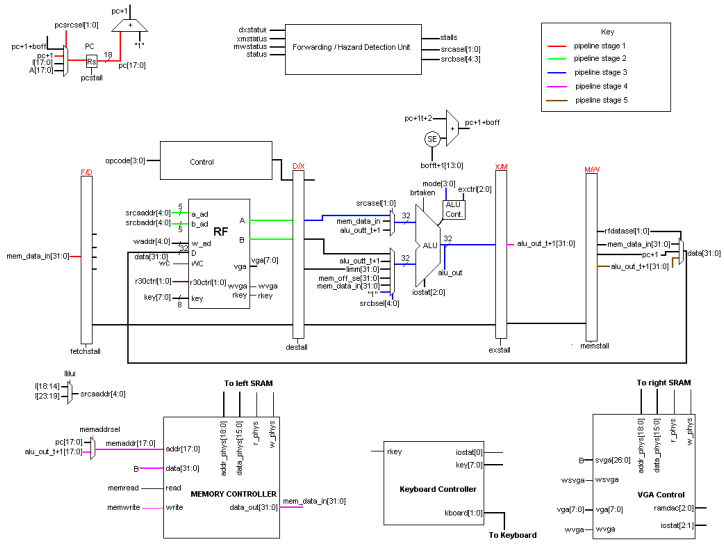
`push`

Opcode 31 : 28	Mode 27 : 24	Destination 23 : 19	SrcA 18 : 14	SrcB 13 : 9	8 : 0
1010	0000	xxxxx	11110	bbbbb	xxxxxxxxx

STACK ← b, $30 ← $30 – 1

```
push $b
```

Push stores the value in register $b to the location in memory specified by register $30. Then register $30 is decremented to move the stack pointer.



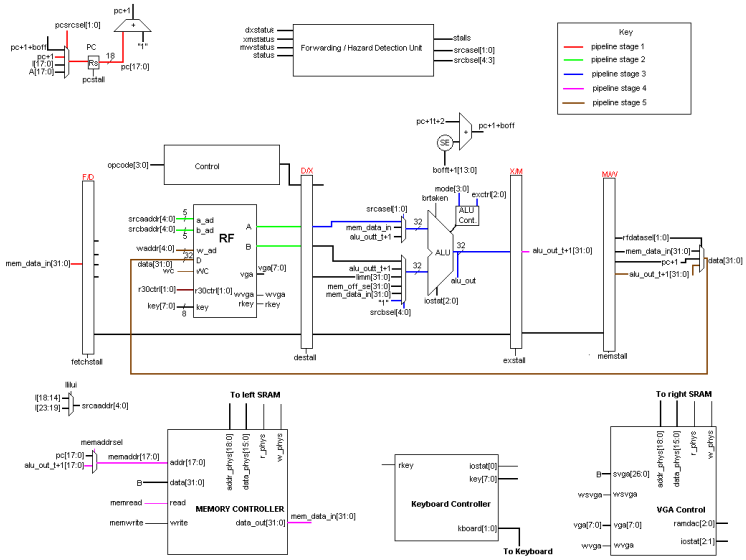
Data flow for `push`.

`pop`

Opcode 31 : 28	Mode 27 : 24	Destination 23 : 19	SrcA 18 : 14	13 : 9	8 : 0
1011	0000	ddddd	11110	xxxxxx	xxxxxxxxx

$d ← STACK
`pop $d`

Pop loads register $d with the value at the address specified by register $30. Then register $30 is incremented.



Data flow for `pop`.

2.8 Jump Instructions

Jump instructions change the flow of execution by loading the PC (program counter) with a new value specified by a register or an immediate value.

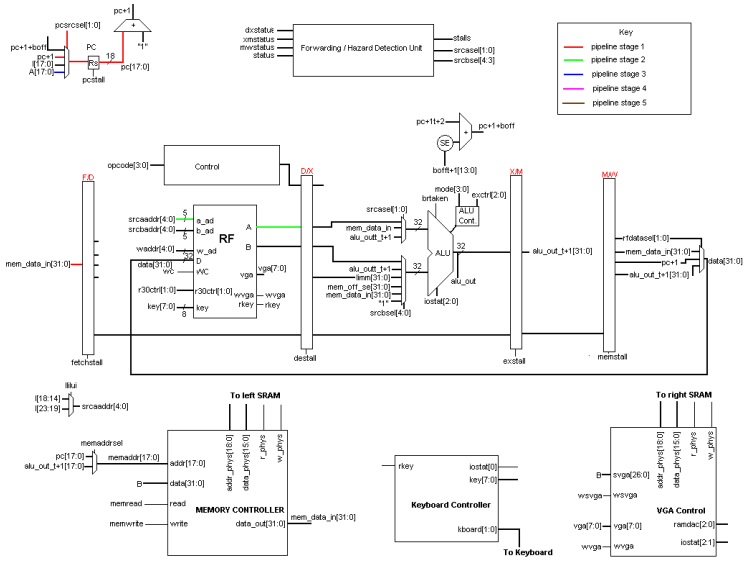
`jr`

Opcode 31 : 28	Mode 27 : 24	 23 : 19	SrcA 18 : 14	 13 : 9	 8 : 0
0111	0000	xxxxx	aaaaa	xxxxx	xxxxxxxxx

PC ← aaaaa

```
jr $a
```

Jump register stores the value in register $a into the PC and begins a new program flow. Jumps must flush the pipeline of any instructions that have begun execution erroneously.



Data flow for `jr`.

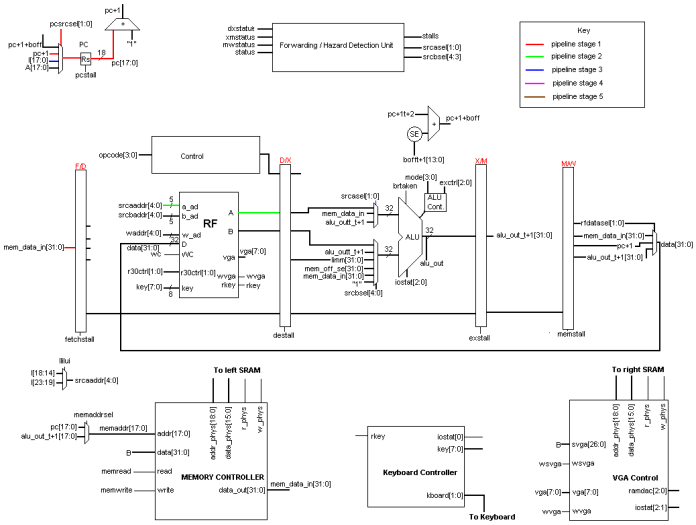
`jal`

Opcode 31 : 28	Mode 27 : 24	Destination 23 : 19	Addr 18 : 14	Addr 13 : 9	Addr 8 : 0
0111	0001	11111	xAAAA	AAAAA	AAAAAAAAA

PC ← AAAAAAAAAAAAAAAAAA

```
jal AAAAAh
```

Jump and link loads the PC with the value specified by the offset A. A can be specified as a label or as a numerical offset. Jumps must flush the pipeline of any instructions that have begun execution erroneously.



Data flow for `jal`.

2.9 Branch Intsructions

Branch instructions change the exectuion flow by loading the PC with the current value of the PC plus the offset field.

`brgt`

Opcode 31 : 28	Mode 27 : 24	Offset 23 : 19	SrcA 18 : 14	SrcB 13 : 9	Offset 8 : 0
0110	0001	ooooo	aaaaa	bbbbb	ooooooooo

PC ← PC + 1 + o

```
brgt $a $b o
```

Branch greater than sets the PC to the value of the PC plus the offset if register $a is greater than register $b.

`brlt`

Opcode 31 : 28	Mode 27 : 24	Offset 23 : 19	SrcA 18 : 14	SrcB 13 : 9	Offset 8 : 0
0110	0010	ooooo	aaaaa	bbbbb	ooooooooo

PC ← PC + 1 + o

```
brlt $a $b o
```

Branch less than sets the PC to the value of the PC plus the offset if register $a is less than register $b.

`breq`

Opcode 31 : 28	Mode 27 : 24	Offset 23 : 19	SrcA 18 : 14	SrcB 13 : 9	Offset 8 : 0
0110	0101	ooooo	aaaaa	bbbbb	oooooooo

PC ← PC + 1 + o
`breq $a $b o`

Branch equal sets the PC to the value of the PC plus the offset if register $a is equal to register $b.

`br`

Opcode 31 : 28	Mode 27 : 24	Offset 23 : 19	18 : 14	13 : 9	Offset 8 : 0
0110	0000	ooooo	xxxxx	xxxxx	oooooooo

PC ← PC + 1 + o
`br o`

Branch sets the PC to the value of the PC plus the offset unconditionally.

`brchar`

Opcode 31 : 28	Mode 27 : 24	Offset 23 : 19	18 : 14	13 : 9	Offset 8 : 0
0110	0011	ooooo	xxxxx	xxxxx	oooooooo

PC ← PC + 1 + o
`brchar o`

Branch character sets the PC to the value of the PC plus the offset if a character is ready to be read from the keyboard.

`brvid`

Opcode 31 : 28	Mode 27 : 24	Offset 23 : 19	18 : 14	13 : 9	Offset 8 : 0
0110	0100	ooooo	xxxxx	xxxxx	oooooooo

PC ← PC + 1 + o
`brvid o`

Branch video sets the PC to the value of the PC plus the offset if the VGA character buffer is ready to be written to.

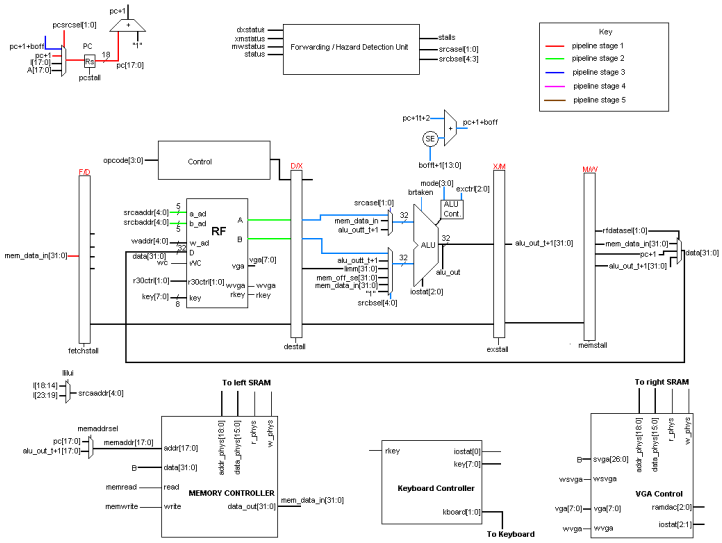
`brpix`

Opcode 31 : 28	Mode 27 : 24	Offset 23 : 19	18 : 14	13 : 9	Offset 8 : 0
0110	0110	ooooo	xxxxx	xxxxx	oooooooo

PC ← PC + 1 + o

`brpix o`

Branch pixel sets the PC to the value of the PC plus the offset if the VGA pixel buffer is ready to be written to.



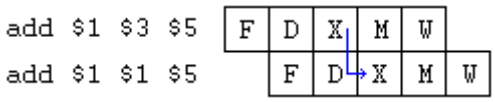
Data flow for branches.

2.10 Data Dependencies

There are several forms of data dependencies that occur in this ISA. For instance

```
add $1 $3 $5
add $1 $1 $5
```

would create a data dependency because the first add instruction would not write to the register file before the second instruction needed the value in register $1. In order to solve this problem, there is a data forwarding line from the MEM stage to the EX stage which provides the ALU with the value before its written to the register file:



4 Software

4.1 Development Software

Several complete programs were produced to aid in the development of RISC-E hardware and software.

4.1.1 `Sim`

`Sim` is a command-line simulator for the RISC-E (RISC – Extended) instruction set. It was written with two purposes in mind:

1) Enable software development for the RISC-E architecture before the architecture is implemented in hardware.
2) Provide a means of testing for the RISC-E architecture by providing a method to debug test programs before execution on the RISC-E processor.

The use of `sim` has allowed parallel development of hardware and software, effectively reducing the time required to generate meaningful programs in the RISC-E instruction set. Test programs and demonstration programs were developed and debugged without concern for potential hardware malfunctions.

`Sim` was written in C/C++ for Win32 machines. Therefore, use of an MS-DOS prompt (a command-line) is required for effective use of `sim`. However, use of batch files (files with a .bat extension in Windows) allows effective invocation of `sim` from the Windows GUI.

Accompanying `sim` is the `bmpgen` program, a bitmap generation utility for displaying pixel changes. The simulator itself does not implement pixel-change information—it instead prints a message to *stdout* or a specified output file that characterizes each individual pixel change. The `bmpgen` program can convert the `sim` output to a meaningful Windows bitmap image (with a .bmp extension), which can in turn be displayed by any graphics program, such as Windows Paint. For further information on `bmpgen` and its syntax, see the appropriate section of this manual.

Effective use of `sim` and `bmpgen` enables programmers to emulate actually running a program on the RISC-E processor, but also provides helpful debugging and tracing tools to speed the development process.

4.1.1.1 Syntax of sim

Correct syntactical usage of sim is essential to using the program effectively. The correct syntax is:

```
sim <input file> [output file] [switches]
```

Note that arguments enclosed in < > are required, but arguments enclosed in [ ] are not, and order of arguments is checked. Thus:

```
sim myfile.asm myfile.out
```

and

```
sim test1.asm -v +r +m
```

Are valid invocations of sim, but

```
sim -v test1.asm
```

is not, as a switch is listed before the input file (`test1.asm`).

Specifying an output file for `sim` has the same effect as redirecting `sim`'s output to a file using the > operator in MS-DOS. Thus:

```
sim stdlib.asm -v +r > stdlib.out
```

and

```
sim stdlib.asm stdlib.out -v +r
```

are equivalent executions of `sim`.

A complete list of switches for `sim`:

Switch	**Description**
`+h`	Display syntax message.
`+r`	Dump register file to *output stream* after execution.
`+m`	Dump non-zero memory locations to *output stream* after execution
`-v`	Verbose mode off—Does not echo code to *output stream.*
`+i`	Instruction Memory Fill—Fills instruction memory with 0xFFFFFFFF to simulate presence of instructions at those memory addresses.
`+vgasilent`	Does not print pixel manipulation information to *output stream*. Useful for diagnosing infinite loops.
`+fast`	Disables long loop waits for I/O (see instruction set and I/O Section below). Recommended only for pixel manipulation-intensive programs.

Invoking `sim` without arguments or with improper arguments will also display the above-mentioned list of arguments and switches.

4.1.1.2 Programming with `sim`

The simulator was designed to emulate a RISC-E architecture's environment as accurately as possible. Therefore, capabilities and limitations that exist in hardware also exist in `sim`, with some exceptions.

The rules of language syntax for `sim` are those one might expect for an assembler—illegal opcodes and registers, undefined labels, and inappropriate offsets will be flagged as errors. E.g.:

```
Command Prompt

C:\Working\sim>sim illegal.asm
// My code begins here
inc  R1        // This instruction is OK
dec  R2        // This instruction is OK
sw   R1  R2  0
add  R5  R1  R2

// The next instruction's opcode is bad
div  R3  R1  R2
*** Error encountered, line 8:
-->Unknown opcode


C:\Working\sim>_
```

The response to an illegal opcodes

In the example above, the first four instructions are valid—the opcodes are defined and the arguments are correctly specified. However, the RISC-E architecture does not include a '`div`' instruction, hence `sim`'s response of "Unknown opcode." Similar responses exist for other errors, such as those listed above. Note that errors in syntax abort the simulation—to continue would be to return ambiguous results.

The memory system in `sim` also faithfully represents the RISC-E architecture—the acceptable addressable range is 0x00000 to 0x3FFFF (an 18-bit range, the logical range of a RISC-E system). **Memory accesses out of the acceptable range are handled by the simulator in the same manner of actual hardware—the upper 14 bits are simply truncated.** Thus, a read from address 0x1247FFFF will read from memory location 0x3FFFF.

It is important to note that `sim` does not assemble or locate code. Therefore, reads from "instruction memory" will not return valid instructions, and nor will writes to "instruction memory" in any way affect execution. The `+i` option exists to fill "instruction memory" with 0xFFFFFFFF, if it is desirable to flag it in this way. In this manner, `sim` is not faithful to the RISC-E architecture—**instruction and data memory are effectively separate in the sim environment**, but not in the true RISC-E system.

4.1.1.3 Instructions in `sim` `<instruction> [arguments]`

The following is a brief listing of recognized instructions in `sim`:

(For a more comprehensive listing, see the RISC-E Programmer's Manual)

Instruction	Arguments	Description (RTL if applicable)
`add`	regd rega regb	Addition: D ← A + B
`sub`	regd rega regb	Subtraction: D ← A – B
`inc`	regd	Increment: D ← D + 1
`dec`	regd	Decrement: D ← D – 1
`mul`	regd rega regb	Multiply: D ← A * B Note: Multiplies lower 16-bits of A and B to produce a 32-bit result.
`and`	regd rega regb	Logical AND: D ← A & B
`or`	regd rega regb	Logical OR: D ← A | B
`not`	regd rega	Logical NOT: D ← ~A
`xor`	regd rega regb	Logical XOR: D ← A ^ B
`lw`	regd regaddr imm_dec9	Mem: D ← memory[regaddr + imm_dec9]
`sw`	rega regaddr imm_dec9	Mem: memory[regaddr + imm_dec9] ← A
`svga`	rega	Pixel manipulation, see I/O Sub-Section
`push`	rega	Stack: TOS ← A, R30 ← R30 – 1
`pop`	regd	Stack: D ← TOS+1, R30 ← R30 + 1
`sl`	regd rega regb	Logical Left Shift: Reg B contains shift amount.
`srl`	regd rega regb	Logical Right Shift: Reg B contains shift amount.
`sra`	regd rega regb	Arithmetic Right Shift: Reg B contains shift amount.
`ror`	regd rega regb	Rotate Right: Reg B contains rotate amount.
`rol`	regd rega regb	Rotate Left: Reg B contains rotate amount.
`lui`	regd imm_hex16	Load Upper Immediate: D ← {UH_D , imm_hex16}
`lli`	regd imm_hex16	Load Lower Immediate: D ← {UH_D , imm_hex16}
`breq`	rega regb label	Branch to label if A = B
`brgt`	rega regb label	Branch to label if A > B
`brlt`	rega regb label	Branch to label if A < B
`brchar`	label	Keyboard polling branch, see I/O Sub-Section
`brpix`	label	Pixel polling branch, see I/O Sub-Section
`brvid`	label	Character polling branch, see I/O Sub-Section
`br`	label	Unconditional branch
`nop`		No-operation

`jr`	regaddr	Jump to value specified by register
`jal`	label	Jump to label (unconditional)

Recognized instructions and their functions

Abbreviation	Definition
rega, A	rega is a source register, A is its content
regb, B	regb is a source register, B is its content
regd, D	regd is the destination register, D is its content (sometimes also used as a source)
regaddr	regaddr is a source register containing an address
imm_dec9	A decimal user-specified immediate value, bounded by: -2^8 < imm_dec9 < 2^8 –1 (2's complement)
imm_hex16	A hexadecimal user-specified immediate value. This may be replaced by a **positive decimal value** if it falls within the range 0x0000 to 0xFFFF. Otherwise, it must be specified as $\mathbf{D_3D_2D_1D_0h}$ where $\mathbf{D_i}$ is a valid hex digit. **Omitting the terminal h may yield undesired results,** as sim will attempt to cast this number as a decimal value.
UH_D	The upper 16 bits of D (D is specified above)
TOS	Top-Of-Stack, or memory[R30]
label	A user-defined label. See Labels below.

A list of abbreviations and their meanings

4.1.1.4 Registers and Arguments

Registers may be specified by any of three methods:

`R<number>` `r<number>` `$<number>`

The range of `<number>` above is **0 – 31**, specified in **decimal**.

There are four special-purpose registers in the RISC-E architecture. They are accessed normally as a general purpose register, but also serve the following purposes:

Register	Remarks
`R0`	Register is always value 0x00000000.
`R29`	Register is dedicated to I/O. See I/O Sub-Section
`R30`	Stack Pointer, also usable as a general-purpose register if not employing a stack.
`R31`	Return-address register. `jal` instructions place the return address in this register—ideally the system will eventually execute `jr R31` to return.

Special purpose registers

Most instructions require one or more arguments. Argument type varies by instruction. The user should consult the tables above for argument requirements of a particular

instruction. Multiple arguments may be delimited by whitespace (excluding the newline character) or commas. The following are all acceptable instructions:

```
add R1 r2 r3
xor $6,  $7,  $8
breq  r1   $0, XLABEL
```

4.1.1.5 Labels

Labels are defined by any line starting with a colon (`:`), followed by up to seven alphanumeric characters. Labels are referenced by their name only—**do not include the colon when referencing a label.** Labels are case-sensitive in `sim`. Thus:

```
:ULAB1
br ULAB1
```

Is an infinite loop, but

```
:ULAB1
br :ULAb1
```

generates an error.

Labels have the following restrictions:

- Labels may not exceed seven characters in length (excluding the colon)
- The colon is only used only to declare a label, not to reference it.
- A label may not begin with any valid hexadecimal character (eg **0**-**9**, **A**-**F**, or **a**-**f**) and may not start with **R**, **r**, or **$**.
- For assembler compatibility, labels may not appear on the same line as comments.

4.1.1.6 Comments and Whitespace

To aid in code readability, comments may be inserted in a source file to highlight key sections or explain complex algorithms. All comments recognized by `sim` begin with a double forward-slash, //. The semantics of the //-type comment are identical to that of popular programming languages, such as C or Java. These comments may be placed anywhere in the source file, with the following exceptions:

- Lines consisting of only comments and whitespace must begin with //, and may not begin with whitespace.
- For assembler compatibility, labels may not appear on the same line as comments.

Note: While C-style // comments are supported by `sim`, block-comments of the /* */ type are not supported.

Whitespace (both horizontal and vertical) may be used arbitrarily by the programmer to improve code readability without affecting execution of `sim`.

4.1.1.7 Reserved word `stop`

`STOP` (case insensitive) is a reserved word in sim. It exists to ensure 100% compatibility with the RISC-E assembler, which requires `stop` at the end of a source file. Should a program in execution encounter `stop`, execution will cease normally.

`STOP` should only be placed at the end of source file—placing `stop` in the middle or beginning of a file will cause a miscalculation of all branch/jump addresses that cross the `stop` reserved word.

4.1.1.8 I/O (Input / Output)

The RISC-E system utilizes a VGA output device and a (PS/2) keyboard input device. The programmer may access these devices to perform I/O in the course of program execution. There are three methods of I/O in the RISC-E architecture: keyboard input, character output, and pixel output. The simulator will directly support both keyboard input and character output, and will support pixel output with the use of the accompanying `bmpgen` program.

Keyboard input:

To read a character from the keyboard, the programmer need only read a value from register `R29`. `R29` is a dedicated I/O register—the lower eight bits of this register represent the last key depressed on the keyboard (in ASCII). If no ASCII code exists for a key, and its code is not otherwise defined in the table below, then the code visible in `R29` will be 00h on a true RISC-E system. In `sim`, its code will be determined by the C *getch*() function call. In this way, `sim` is not true to the RISC-E architecture, and thus the user should only attempt to use `sim` when expecting defined inputs.

Key	**`R29` Code**
↑	38h
↓	32h
←	34h
→	36h

Key	**`R29` Code**
RETURN	0Ah
ENTER	0Ah

Nonstandard key codes

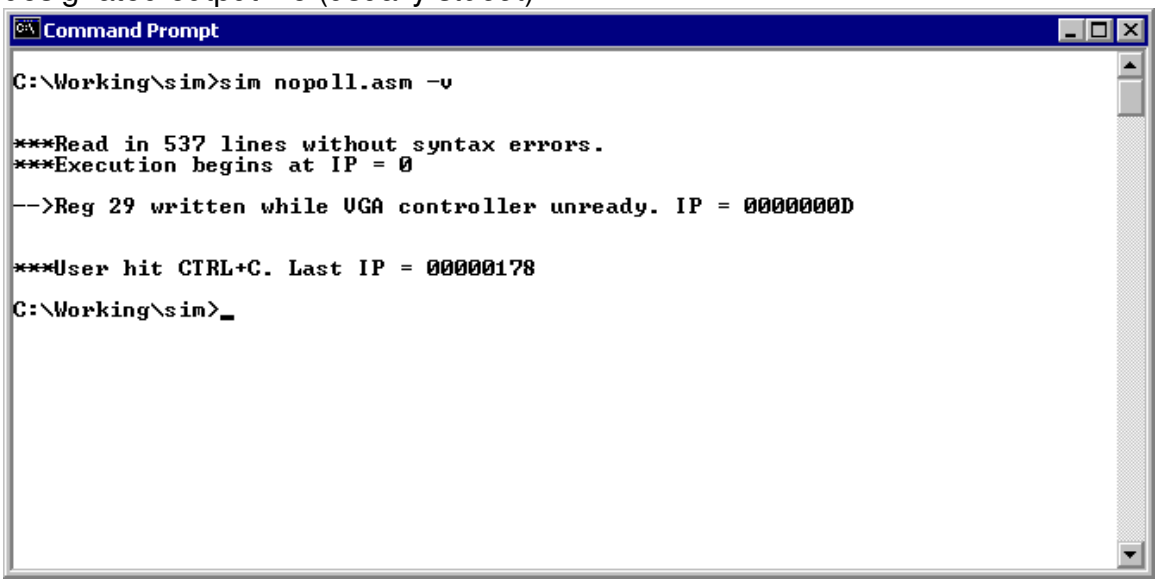
It is important to note that `R29` may be read at any time, though its value may not always be meaningful. If the user has not depressed any key, then the value of `R29` should not be considered valuable input. Thus, the programmer must first poll the keyboard interface using the `brchar` instruction to determine if a key has been pressed. (`brchar` is taken if the value in `R29` has not yet been read.)

To simulate this behavior, `sim` actually implements a counter that is initially random and decrements with each successive `brchar` instruction. When this counter reaches zero, `sim` will initiate a key request calling C's *getch*() function. The *getch*() function returns the pressed character (without echoing to the screen) and that value is then stored in `R29`, presumably to be read in subsequent execution.

As mentioned above, use of *getch*() limits `sim`'s ability to correctly emulate the RISC-E keyboard interface. However, for most user inputs `sim`'s performance matches that of hardware. Alphanumerics, whitespace, and most symbols will function identically in RISC-E and in `sim`.

Character Output:
Writing a character to the screen in the RISC-E system is equally as simple as reading from the keyboard. Again, register `R29` is used to interface with the I/O system. Writes to `R29` will initiate character output, if the VGA controller is ready to accept a new character. The value written into `R29` will be truncated to its lower eight bits, which will be interpreted as an ASCII character by the VGA controller. For polling purposes, branch instruction `brvid` will be taken when the VGA controller standing by for a new character.

As in the case of character input, it is highly recommended that the programmer make careful use of the `brvid` instruction to ensure that `R29` is written only when the VGA controller is ready. Failing to poll correctly in hardware will result in the character write request to be ignored. Failing to do so in `sim` will generate a warning printed to the designated output file (usually *stdout*):



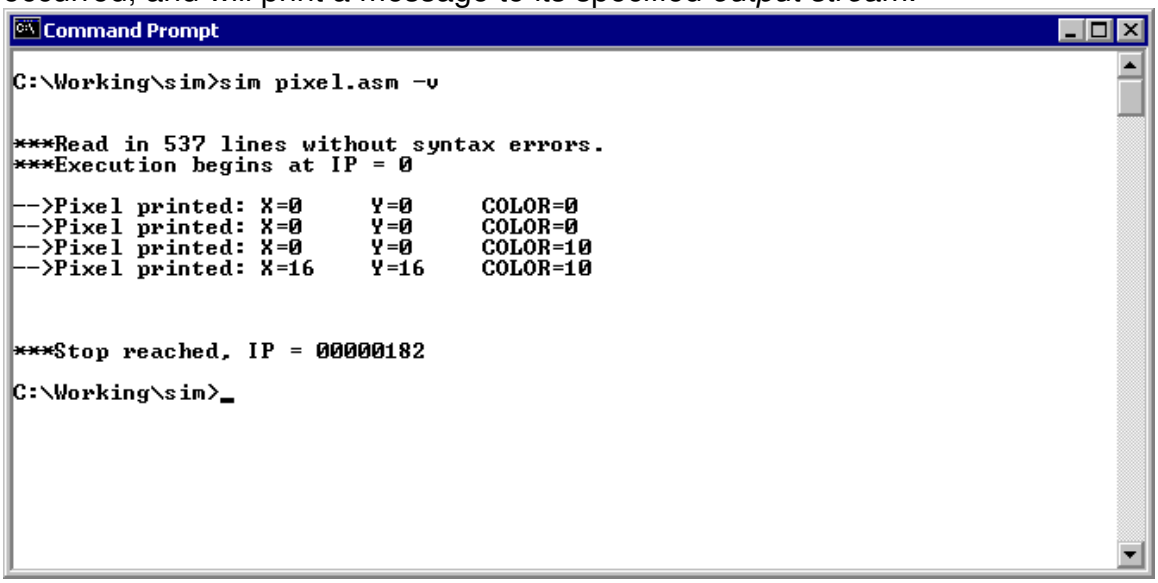
Response to a non-polled `R29` write

Character output in `sim` employs the C function *putc*(). As a result, `sim` also is unable to exactly match the character-output behavior of the RISC-E system, as *putc*() will print some symbols that are not recognized in the RISC-E architecture. Additionally, the VGA controller can also accept commands via the `R29` interface that cannot be implemented in `sim` (for details, consult the VGA Controller documentation). Finally, MS-DOS command-lines will scroll output as more and more characters are printed to the screen, but there will is no such scrolling action in the RISC-E character output system—instead the user must erase the screen by writing backspace characters followed by space characters. The backspace/space requirement is modeled correctly in `sim`.

Pixel output:
The RISC-E architecture also allows the user to perform individual pixel manipulations. As in the case of character output and keyboard input, a polling instruction exists to facilitate timing of pixel manipulation requests. This instruction is `brpix`—it is used identically to that of `brvid` and `brchar`. Branch `brpix` is taken if the VGA controller is ready to accept a pixel manipulation.

Unlike character I/O, RISC-E affords a separate instruction to pixel manipulation, the `svga` instruction. Note that `svga` denotes "send to VGA," and does not reference the common acronym SVGA. The `svga` instruction takes one register as its argument—the pixel manipulation data is entirely encapsulated in that register. To reference how to format pixel data, refer to the RISC-E Programmer's Manual or the VGA Controller documentation.

Pixel manipulation is not completely supported by `sim`, but the `bmpgen` program can be used to render `sim`'s pixel-manipulation output into a viewable format (see 4.1.1.9 `bmpgen`). However, `sim` will recognize when a successful pixel manipulation has occurred, and will print a message to its specified *output stream*:



Response to four pixel outputs

For programs that are pixel-manipulation intensive, it is recommended that the `+vgasilent` and `+fast` command-line options be employed. When `sim` is invoked with these switches, no pixel manipulation information will be printed to the specified *output stream*, and the polling requirements of the simulated output device will be relaxed substantially. Execution time for these programs will be dramatically reduced.

4.1.1.9 bmpgen for Viewing Pixel Output

It is possible to view pixel manipulations generated by a program once execution has terminated by using the bmpgen program. bmpgen will convert the pixel manipulation data generated by sim into a viewable (bitmap, *.bmp) format.

The output bitmap always has width 256 and height 256 (pixels). Pixel manipulations outside of these dimensions are not visible using bmpgen. The background color of this bitmap is black—therefore writing black-colored pixels will not be visible after using bmpgen. The colors that are generated by bmpgen are in general not the same colors that will appear on a RISC-E system—bmpgen is intended to show pixel patterns, not true pixel colors. However, the colors white (FFh or 255 decimal) and black (00h or 0 decimal) are accurately modeled in bmpgen.

To syntax of bmpgen is:

```
bmpgen <input file> <output file>
```

eg

```
bmpgen pixels.out pixels.bmp
```

Both the <input file> and <output file> arguments are required. The result of the execution (in <output file>) is a bitmap image. The <input file> format is ASCII text, though it is expected that this file will be an output file from an invocation of sim. The bmpgen program will examine this file and look for pixel-manipulation statements like those shown in the example above. It will then modify the output bitmap accordingly. Therefore, only the most recent update to a pixel will be visible.

Note: The +vgasilent switch should not be used when generating an input file for bmpgen. However, the use of +fast is highly recommended.

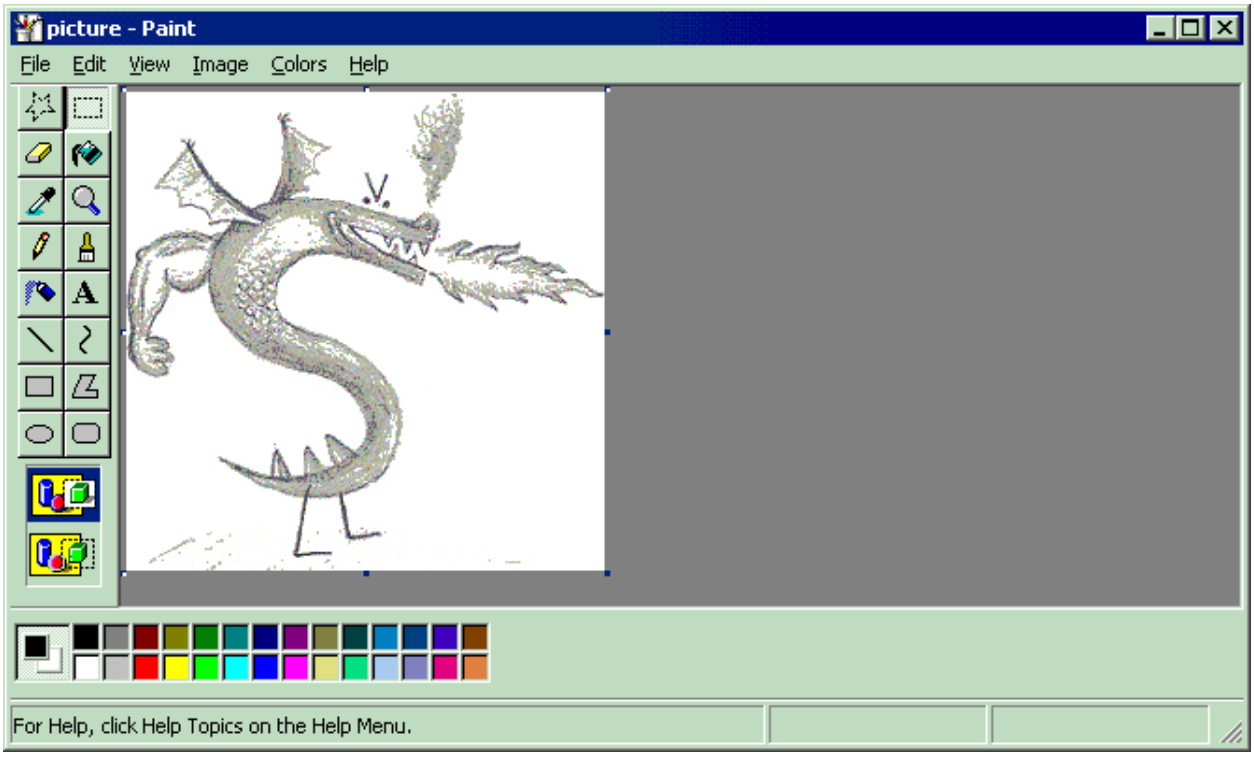
The following is a complete example of how to use `bmpgen`:

```
Command Prompt

C:\Working\sim>sim picture.asm picture.out -v +fast
Using output file: picture.out

C:\Working\sim>bmpgen picture.out picture.bmp
Detected 65537 pixel writes.

C:\Working\sim>picture.bmp

C:\Working\sim>_
```

invoke sim with output file specified

run bmpgen on output file to produce bitmap

invoke Paint to view the bitmap



4.1.1.10 Files

`sim.exe`	The executable form of `sim`
`bmpgen.exe`	The executable form of `bmpgen`
`header`	A required data file for `bmpgen`
`sim.cpp`	Source code for `sim`
`instrs.inc`	Source code for `sim`
`stdlib.asm`	A collection of useful functions
`dobitmap.bat`	An MS-DOS batch shell for streamlining use of `sim`+`bmpgen`