r32 CPU and Instruction Set

Reed Foster

June 2017

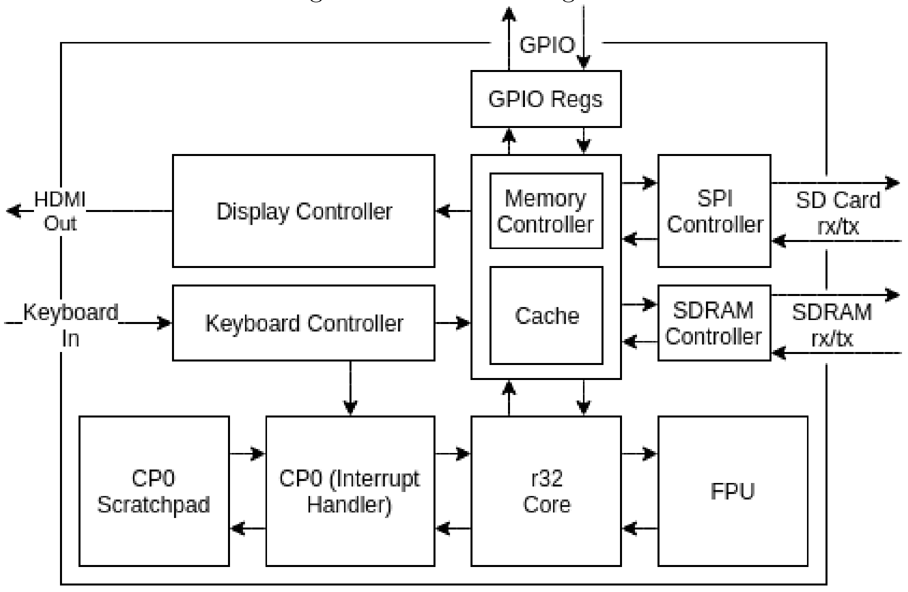
Contents

About r32

r32 is a fully-fledged, 32-bit microcontroller/SoC with a CPU-core based on the MIPS architecture.

Architecture

Figure 1: r32 Block Diagram



1 Instruction Set

The r32 core instruction set is based on MIPS, an instruction set developed MIPS Technologies (founded by researchers from Stanford University). MIPS is a RISC architecture, with a 5-stage pipeline and 128-byte register file. r32's instruction set is mostly a subset of MIPS, with a few tweaks to the programmer's interface to the processor.

1.1 Instruction Format

Just like MIPS, r32 has three different instruction formats: r-, i-, and j-type instructions (and several other formats for floating point instructions, which are based off of r-type instructions). R-type instructions are primarily computational instructions, while i-type instructions are a mix of memory access, branch, and computation instructions, and j-type instructions are used for setting the program counter to a specific value (like an absolute branch rather than a relative one). The opcode of the instruction determines its type/format, as well as what the processor does. Registers rs and rt are source registers for r-type instructions, and rd is the destination register. For loads and stores, rs is used to determine the RAM address, while rt is either the destination or source (for load or store, respectively).

1.1.1 R-Type Instructions

opcode	rs	rt	rd	shamt	func
31 26	25 21	20 16	15 11	10 6	5 0

1.1.2 I-Type Instructions

opcode	rs	rt	imm
31 26	25 21	20 16	15 0

1.1.3 J-Type Instructions

opcode	jaddr
31 26	25 0

1.2 Processor Instructions

r32 supports a wide variety of instructions which are listed in Table 1.

CPU Core		
add	addi	addiu
addu	and	andi
beq	bgez	bgtz
blez	bltz	bne
j	jr	lb
lbu	lh	lhu
lui	lw	nor
or	ori	pref
sb	sh	sll
sllv	slt	slti
sltiu	sltu	sra
srav	srl	srlv
sub	subu	sw
syscall	teq	teqi
tge	tgei	tgeiu
tgeu	tlt	tlti
tltiu	tltu	tne
tnei	xor	xori

CP0	
mfc0	mtc0

CP1 (FPU)		
bc1f	bc1t	ceq
cfc1	cle	clt
ctc1	fabs	fadd
fdiv	fma	fmul
fneg	fsqrt	fsub
lwc1	mfc1	mov
mtc1	swc1	

Table 1: Supported processor and coprocessor instructions

add

000000	rs	rt	rd	00000	100000
31 26	25 21	20 16	15 11	10 6	5 0

Add instruction (trap on overflow)
Reg[rd] $\leftarrow$ Reg[rs] + Reg[rt]

addi

001000	rs	rt	imm
31 26	25 21	20 16	15 0

Add immediate (trap on overflow)
Reg[rt] $\leftarrow$ Reg[rs] + imm$^{\pm}$

addiu

001001	rs	rt	imm
31 26	25 21	20 16	15 0

Add immediate (ignore overflow, note imm is still sign extended)
Reg[rt] $\leftarrow$ Reg[rs] + imm$^{\pm}$

addu

000000	rs	rt	rd	00000	100001
31 26	25 21	20 16	15 11	10 6	5 0

Add instruction (ignore overflow)

Reg[rd] $\leftarrow$ Reg[rs] + Reg[rt]

and

000000	rs	rt	rd	00000	100100
31 26	25 21	20 16	15 11	10 6	5 0

Bitwise logical AND

Reg[rd] $\leftarrow$ Reg[rs] AND Reg[rt]

andi

001100	rs	rt	imm
31 26	25 21	20 16	15 0

Bitwise logical AND with immediate

Reg[rt] $\leftarrow$ Reg[rs] AND imm$^{\emptyset}$

bc1f

010001	01000	cc	0	0	imm
31 26	25 21	20 16	15	0	

Branch if coprocessor 1 (FPU) false

PC $\leftarrow$ (cc = 0) ? PC + offset$^{\pm}$: PC + 1

bc1t

010001	01000	cc	0	1	imm
31 26	25 21	20 16	15	0	

Branch if coprocessor 1 (FPU) true

PC $\leftarrow$ (cc = 1) ? PC + offset$^{\pm}$: PC + 1

beq

000100	rs	rt	offset
31 26	25 21	20 16	15 0

Branch if equal

PC $\leftarrow$ (rs = rt) ? PC + offset$^{\pm}$: PC + 1

bgez

000001	rs	0001	offset
31 26	25 21	20 16	15 0

Branch if greater than or equal to zero

PC $\leftarrow$ (rs $\geq$ 0) ? PC + offset$^{\pm}$: PC + 1

bgtz

000111	rs	0000	offset
31 26	25 21	20 16	15 0

Branch if greater than zero

PC ← (rs > 0) ? PC + offset$^{\pm}$: PC + 1

blez

000110	rs	0000	offset
31 26	25 21	20 16	15 0

Branch if less than or equal to zero

PC ← (rs $\leq$ 0) ? PC + offset$^{\pm}$: PC + 1

bltz

000001	rs	0000	offset
31 26	25 21	20 16	15 0

Branch if less than zero

PC ← (rs < 0) ? PC + offset$^{\pm}$: PC + 1

bne

000101	rs	rt	offset
31 26	25 21	20 16	15 0

Branch if less not equal

PC ← (rs $\neq$ rt) ? PC + offset$^{\pm}$: PC + 1

ceq

010001	10000	ft	fs	cc	00	11	0010
31 26	25 21	20 16	15 11	10 8	7 6	5 4	3 0

Floating point compare (equal)

CCR[cc] ← (fs = ft) ? 1 : 0

cfc1

010001	00010	rt	fs	00000000000
31 26	25 21	20 16	15 11	10 0

Move from floating point (note fs must be 0 or 31)

Reg[rt] ← FCR[fs]

cle

010001	10000	ft	fs	cc	00	11	1110
31 26	25 21	20 16	15 11	10 8	7 6	5 4	3 0

Floating point compare (less than or equal to)

CCR[cc] ← (fs $\leq$ ft) ? 1 : 0

clt

010001	10000	ft	fs	cc	00	11	1100
31 26	25 21	20 16	15 11	10 8	7 6	5 4	3 0

Floating point compare (less than)
CCR[cc] $\leftarrow$ (fs $<$ ft) ? 1 : 0

ctc1

010001	00110	rt	fs	00000000000
31 26	25 21	20 16	15 11	10 0

Move from floating point (note fs = 0 or fs = 31 are the only valid arguments for fs)
FCR[fs] $\leftarrow$ Reg[rt]

fabs

010001	10000	00000	fs	fd	000101
31 26	25 21	20 16	15 11	10 6	5 0

Floating point absolute value
FReg[fd] $\leftarrow$ |FReg[fs]|

fadd

010001	10000	ft	fs	fd	000000
31 26	25 21	20 16	15 11	10 6	5 0

Floating point add
FReg[fd] $\leftarrow$ FReg[fs] + FReg[ft]

fdiv

010001	10000	ft	fs	fd	000011
31 26	25 21	20 16	15 11	10 6	5 0

Floating point divide
FReg[fd] $\leftarrow$ FReg[fs] / FReg[ft]

fma

010001	10000	ft	fs	fd	010111
31 26	25 21	20 16	15 11	10 6	5 0

Floating point multiply add
FReg[fd] $\leftarrow$ Freg[fd] + FReg[fs] * FReg[ft]

fmul

010001	10000	ft	fs	fd	000010
31 26	25 21	20 16	15 11	10 6	5 0

Floating point multiply

FReg[fd] ← FReg[fs] * FReg[ft]

fneg

010001	10000	00000	fs	fd	000111
31 26	25 21	20 16	15 11	10 6	5 0

Floating point negate
FReg[fd] ← -FReg[fs]

fsqrt

010001	10000	00000	fs	fd	000100
31 26	25 21	20 16	15 11	10 6	5 0

Floating point square root
FReg[fd] ← $\sqrt{\text{FReg[fs]}}$

fsub

010001	10000	ft	fs	fd	000001
31 26	25 21	20 16	15 11	10 6	5 0

Floating point subtract
FReg[fd] ← FReg[fs] - FReg[ft]

j

000010	jaddr
31 26	25 0

Jump
PC ← PC (31 downto 28) & jaddr & ”00”

jr

000000	rs	000000000000000	001000
31 26	25 21	20 6	5 0

Jump register
PC ← Reg[rs]

lb

100000	base	rt	offset
31 26	25 21	20 16	15 0

Load byte
Reg[rt] ← Mem[rs + imm$^{\pm}$]$^{\pm}$

lbu

100100	base	rt	offset
31 26	25 21	20 16	15 0

Load byte unsigned

Reg[rt] $\leftarrow$ Mem[rs + imm$^{\pm}$]$^{\emptyset}$

lh

100001	base	rt	offset
31 26	25 21	20 16	15 0

Load halfword

Reg[rt] $\leftarrow$ Mem[rs + imm$^{\pm}$]$^{\pm}$

lhu

100101	base	rt	offset
31 26	25 21	20 16	15 0

Load halfword unsigned

Reg[rt] $\leftarrow$ Mem[rs + imm$^{\pm}$]$^{\emptyset}$

lui

001111	00000	rt	imm
31 26	25 21	20 16	15 0

Load upper immediate

Reg[rt] $\leftarrow$ imm & (15 downto 0 =¿ '0')

lw

100011	base	rt	offset
31 26	25 21	20 16	15 0

Load word

Reg[rt] $\leftarrow$ Mem[Reg[base] + imm$^{\pm}$]

lwc1

100011	base	rt	offset
31 26	25 21	20 16	15 0

Load word to Coprocessor 1 (FPU)

FReg[ft] $\leftarrow$ Mem[Reg[base] + imm$^{\pm}$]

mfc0

010000	00000	rt	rd	00000000	sel
31 26	25 21	20 16	15 11	10 3	2 0

Move from Coprocessor 0

Reg[rt] $\leftarrow$ CP0Reg[rd, sel]

mfc1

010001	00000	rt	fs	00000000000
31 26	25 21	20 16	15 11	10 0

Move from Coprocessor 1 (Floating point)
Reg[rt] ← FReg[fs]

mov

010001	10000	00000	fs	fd	000110
31 26	25 21	20 16	15 11	10 6	5 0

Floating point move FReg[fd] ← FReg[fs]

mtc0

010000	00100	rt	rd	00000000	sel
31 26	25 21	20 16	15 11	10 3	2 0

Move to Coprocessor 0
CP0Reg[rd, sel] ← Reg[rt]

mtc1

010001	00100	rt	fs	00000000000
31 26	25 21	20 16	15 11	10 0

Move to Coprocessor 1 (Floating point)
FReg[fs] ← Reg[rt]

nor

000000	rs	rt	rd	00000	100111
31 26	25 21	20 16	15 11	10 6	5 0

Bitwise logical NOR
Reg[rd] ← Reg[rs] NOR Reg[rt]

or

000000	rs	rt	rd	00000	100101
31 26	25 21	20 16	15 11	10 6	5 0

Bitwise logical OR
Reg[rd] ← Reg[rs] OR Reg[rt]

ori

001101	rs	rt	imm
31 26	25 21	20 16	15 0

Bitwise logical OR with immediate
Reg[rd] ← Reg[rs] OR $\text{imm}^{\emptyset}$

pref

110011	base	00000	offset
31 26	25 21	20 16	15 0

Prefetch
Transfer data from RAM to cache

sb

101000	base	rt	offset
31 26	25 21	20 16	15 0

Store byte
Mem[Reg[base] + offset$^\pm$] $\leftarrow$ Reg[rt]

sh

101001	base	rt	offset
31 26	25 21	20 16	15 0

Store halfword
Mem[Reg[base] + offset$^\pm$] $\leftarrow$ Reg[rt]

sll

000000	00000	rt	rd	shamt	000000
31 26	25 21	20 16	15 11	10 6	5 0

Logical left shift
Reg[rd] $\leftarrow$ Reg[rt] ¡¡ shamt

sllv

000000	rs	rt	rd	00000	000100
31 26	25 21	20 16	15 11	10 6	5 0

Logical left shift (variable)
Reg[rd] $\leftarrow$ Reg[rt] ¡¡ Reg[rs]

slt

000000	rs	rt	rd	00000	101010
31 26	25 21	20 16	15 11	10 6	5 0

Set if less than (signed comparison)
Reg[rd] $\leftarrow$ Reg[rs] $<$ Reg[rt] ? 1 : 0

slti

001011	rs	rt	imm
31 26	25 21	20 16	15 0

Set if less than with immediate (signed comparison)
Reg[rt] $\leftarrow$ Reg[rs] $<$ imm$^\pm$? 1 : 0

sltiu

001011	rs	rt	imm
31 26	25 21	20 16	15 0

Set if less than with immediate (unsigned comparison, note imm is still sign-extended)

Reg[rt] $\leftarrow$ Reg[rs] $<$ imm$^{\pm}$? 1 : 0

sltu

000000	rs	rt	rd	00000	101010
31 26	25 21	20 16	15 11	10 6	5 0

Set if less than (unsigned comparison)

Reg[rd] $\leftarrow$ Reg[rs] $<$ Reg[rt] ? 1 : 0

sra

000000	00000	rt	rd	shamt	000011
31 26	25 21	20 16	15 11	10 6	5 0

Arithmetic right shift

Reg[rd] $\leftarrow$ Reg[rt] ¿¿ shamt

srav

000000	rs	rt	rd	00000	000111
31 26	25 21	20 16	15 11	10 6	5 0

Arithmetic right shift (variable)

Reg[rd] $\leftarrow$ Reg[rt] ¿¿ Reg[rs]

srl

000000	00000	rt	rd	shamt	000010
31 26	25 21	20 16	15 11	10 6	5 0

Logical right shift

Reg[rd] $\leftarrow$ Reg[rt] ¿¿¿ shamt

srlv

000000	rs	rt	rd	00000	000110
31 26	25 21	20 16	15 11	10 6	5 0

Logical right shift (variable)

Reg[rd] $\leftarrow$ Reg[rt] ¿¿¿ Reg[rs]

sub

000000	rs	rt	rd	00000	100010
31 26	25 21	20 16	15 11	10 6	5 0

Subtract (trap on overflow)

Reg[rd] $\leftarrow$ Reg[rs] - Reg[rt]

subu

000000	rs	rt	rd	00000	100011
31 26	25 21	20 16	15 11	10 6	5 0

Subtract (ignore overflow)
Reg[rd] $\leftarrow$ Reg[rs] - Reg[rt]

sw

101011	base	rt	offset
31 26	25 21	20 16	15 0

Store word
Mem[Reg[base] + offset$^\pm$] $\leftarrow$ Reg[rt]

swc1

101011	base	rt	offset
31 26	25 21	20 16	15 0

Store word from Coprocessor 1 (FPU)
Mem[Reg[base] + offset$^\pm$] $\leftarrow$ FReg[ft]

syscall

000000	code	001100
31 26	25 6	5 0

System call
Causes system call exception, tranferring control to exception handler. The `code` field is ignored by hardware but can be retrieved by the exception handler from the instruction stored in memory

teq

000000	rs	rt	code	110100
31 26	25 21	20 16	15 6	5 0

Trap if equal
Traps if Reg[rs] = Reg[rt]. The `code` field is ignored by hardware but can be retrieved by the exception handler from the instruction stored in memory

teqi

000001	rs	01100	imm
31 26	25 21	20 16	15 0

Trap if equal with immediate
Traps if Reg[rs] = imm$^\pm$. The `code` field is ignored by hardware but can be retrieved by the exception handler from the instruction stored in memory

tge

000000	rs	rt	code	110000
31 26	25 21	20 16	15 6	5 0

Trap if greater than or equal to
Traps if Reg[rs] $\geq$ Reg[rt]. The `code` field is ignored by hardware but can be retrieved by the exception handler from the instruction stored in memory

tgei

000001	rs	01000	imm
31 26	25 21	20 16	15 0

Trap if greater than equal to with immediate
Traps if Reg[rs] $\geq$ imm$^{\pm}$. The `code` field is ignored by hardware but can be retrieved by the exception handler from the instruction stored in memory

tgeiu

000001	rs	01001	imm
31 26	25 21	20 16	15 0

Trap if greater than equal to with immediate (unsigned comparison, note imm is still sign-extended)
Traps if Reg[rs] $\geq$ imm$^{\pm}$. The `code` field is ignored by hardware but can be retrieved by the exception handler from the instruction stored in memory

tgeu

000000	rs	rt	code	110001
31 26	25 21	20 16	15 6	5 0

Trap if greater than or equal to (unsigned comparison)
Traps if Reg[rs] $\geq$ Reg[rt]. The `code` field is ignored by hardware but can be retrieved by the exception handler from the instruction stored in memory

tlt

000000	rs	rt	code	110010
31 26	25 21	20 16	15 6	5 0

Trap if less than
Traps if Reg[rs] $<$ Reg[rt]. The `code` field is ignored by hardware but can be retrieved by the exception handler from the instruction stored in memory

tlti

000001	rs	01010	imm
31 26	25 21	20 16	15 0

Trap if less than with immediate

Traps if Reg[rs] $<$ imm$^{\pm}$. The `code` field is ignored by hardware but can be retrieved by the exception handler from the instruction stored in memory

tltiu

000001	rs	01011	imm
31 26	25 21	20 16	15 0

Trap if less than with immediate (unsigned comparison, note imm is still sign-extended)
Traps if Reg[rs] $<$ imm$^{\pm}$. The `code` field is ignored by hardware but can be retrieved by the exception handler from the instruction stored in memory

tltu

000000	rs	rt	code	110011
31 26	25 21	20 16	15 6	5 0

Trap if less than (unsigned comparison)
Traps if Reg[rs] $<$ Reg[rt]. The `code` field is ignored by hardware but can be retrieved by the exception handler from the instruction stored in memory

tne

000000	rs	rt	code	110110
31 26	25 21	20 16	15 6	5 0

Trap if not equal
Traps if Reg[rs] $\neq$ Reg[rt]. The `code` field is ignored by hardware but can be retrieved by the exception handler from the instruction stored in memory

tnei

000001	rs	01110	imm
31 26	25 21	20 16	15 0

Trap if not equal with immediate
Traps if Reg[rs] $\neq$ imm$^{\pm}$. The `code` field is ignored by hardware but can be retrieved by the exception handler from the instruction stored in memory

xor

000000	rs	rt	rd	00000	100110
31 26	25 21	20 16	15 11	10 6	5 0

Bitwise logical XOR
Reg[rd] $\leftarrow$ Reg[rs] XOR Reg[rt]

xori

001110	rs	rt	imm
31 26	25 21	20 16	15 0

Bitwise logical AND with immediate
Reg[rt] $\leftarrow$ Reg[rs] XOR imm$^{\emptyset}$

2 Memory

Memory information

2.1 SDRAM

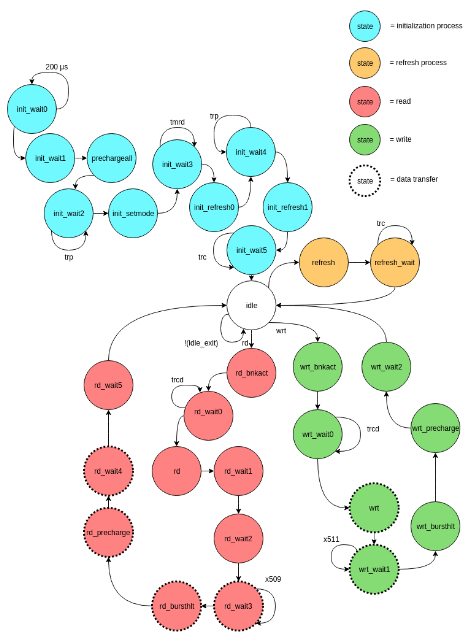
2.1.1 What is SDRAM?

SDRAM stands for Synchronous DRAM, or Synchronous Dynamic Random Access Memory. SRAM, or Static Random Access Memory, the fast style of memory found in cache memory on the CPU die, is made up of transistors that form a circuit that has similar behavior to a latch or flip-flop. DRAM and SDRAM, however, use small capacitors to store each bit, and as a result, DRAM-style memories are much cheaper per byte than SRAM. Because DRAM uses capacitors to store data, each bit must be refreshed frequently to maintain its state. Fortunately, SDRAM modules have control circuitry that automates a lot of the maintainance tasks surrounding DRAM, so memory controllers that interface with SDRAM aren't as difficult to design. That being said, SDRAM presents a far more complex interface than SRAM. One of the first confusing things for beginners is how SDRAM can have millions of addresses, despite a 12- or 13-bit address ($2^{13} = 8192$ or $8K$). In order to access a word of data in SDRAM, one must first activate a row in a bank, and then supply a column address to access a byte when reading. This essentially doubles the 12 or 13 bits of address, allowing for the user to access millions of separate addresses. One of the neat things about SDRAM that allows it to perform very well despite large latencies (which increase at higher clock frequencies) is its capability to burst. The particular SDRAM on the minispartan6+ development board used to develop the r32 allows for 1-word, 2-word, 4-word, 8-word, and full-page (512-word) bursts. Because the r32 specifies a caching hierarchy, the memory controller uses a 512-word burst to maximize bandwith. By only reading or writing to RAM when a cache miss occurs, the controller logic is greatly simplified.

2.1.2 Finite State Machine

At the heart of the SDRAM controller is a Finite State Machine, or FSM. Figure 2 shows the 2nd (current) revision of the SDRAM state machine used to drive the controller. The green states indicate states in which data is being transferred (read/written). First, the SDRAM is initialized (blue states). The datasheet for the SDRAM chip specifies a 200μs startup where all control signals must be held constant. This is achieved by using a counter that counts from $200,000/t_{ck}$ to 0 where t_{ck} is the period of the clock (ns). The rest of the delays are implemented the same way. The loopback state transitions labeled t_{rp}, t_{mrd}, t_{rp}, etc. are timing constraints specified in the SDRAM datasheet.

Figure 2: SDRAM Finite State Machine



2.1.3 FIFOs

In order to maximize cache efficiency, I have added FIFO queues to the data input and output on the SDRAM controller. This allows the cache to quickly free up a line that has old data in it by writing the evicted data to a buffer without waiting for the controller to be ready to write. The FIFO on the output of the controller achieves a similar goal, alllowing the SDRAM controller to read from the SDRAM uninterrupted and write all of the read data to the buffer. In addition to buffering data, I also use a small FIFO as a "request" queue. This allows the CPU cache logic to make requests to read or write to SDRAM, even if the SDRAM controller is currently accessing the SDRAM. The CPU cache can request to write data and begin to write data into the transmit FIFO while the SDRAM controller is still busy processing a previous read or write request, and then proceed to manage the cache until it the data is ready. Originally, all FIFO queues were synchronous, with a single clock for both enqueue and dequeue operations. However, I decided that in order to test that the burst was working properly, I should add dual-clock or asynchronous FIFOs so that I can read data out very slowly (i.e. a human can see the contents of each memory location with 8 leds) from the data out queue.

3 I/O

I/O information