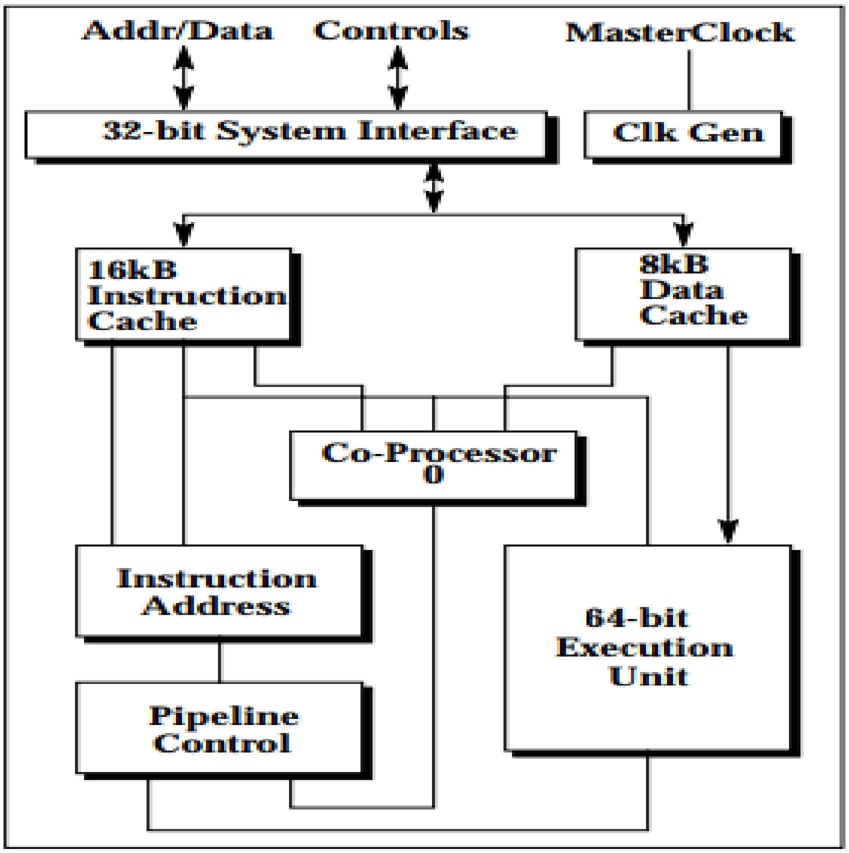
RISC Processor

MISC R4300i

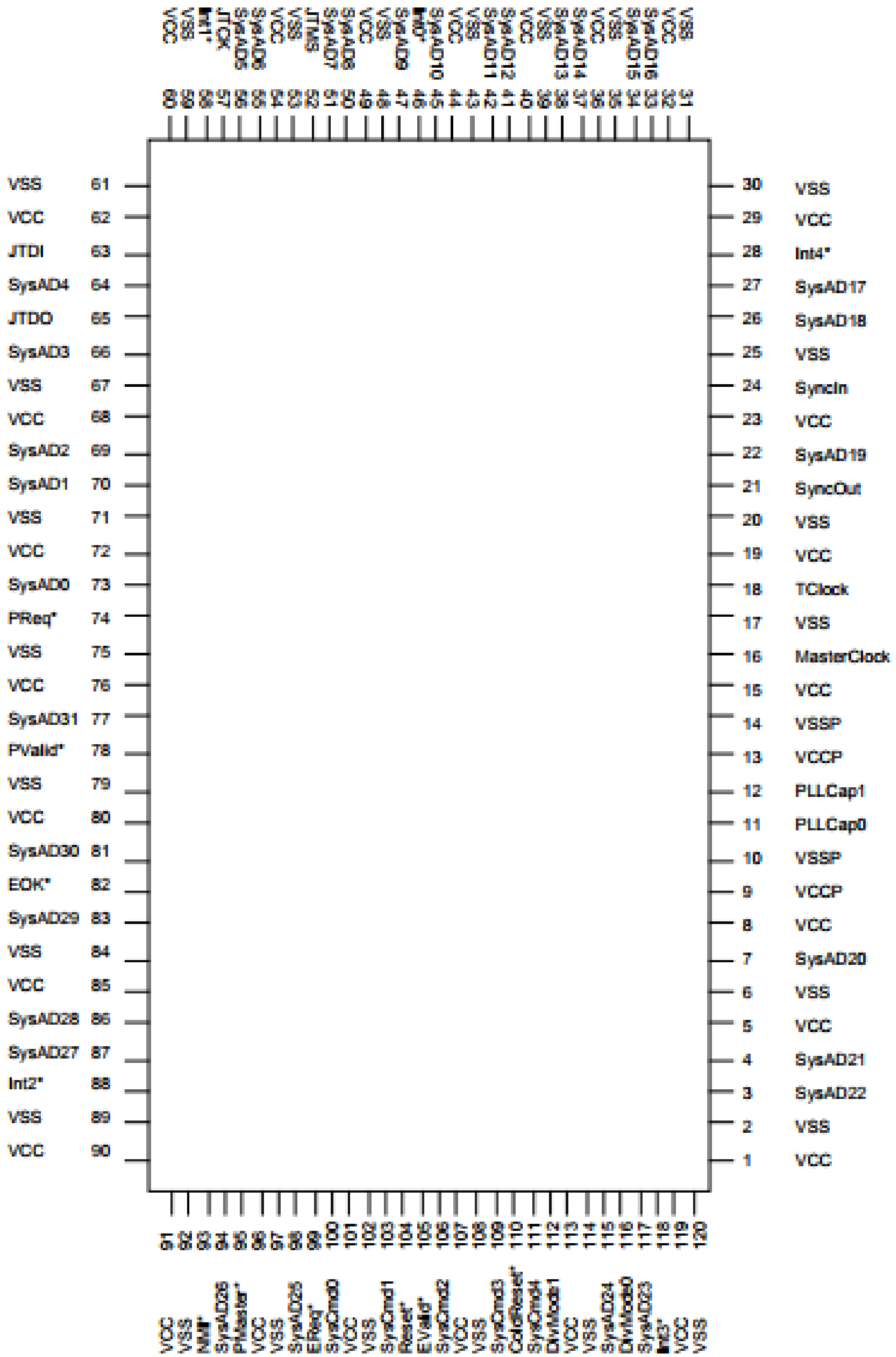
Introduction

The R4300i is a low-cost RISC microprocessor optimized for demanding consumer applications. The R4300i provides performance equivalent to a high-end PC at a cost point to enable set-top terminals, games and portable consumer devices. The R4300i is compatible with the MIPS R4000 family of RISC microprocessors and will run all existing MIPS software. Unlike its predecessors designed for use in workstations, the R4300i is expected to lower the cost of systems in which it is used, a requirement for price-sensitive consumer products. The R4300i is also an effective embedded processor, supported by currently available development tools and providing very high performance at a low price-point.

Block Diagram



Pin Diagram


VCC 60
VSS 59
Int1* 58
JTCK 57
SysAD5 56
SysAD6 55
VCC 54
VSS 53
JTMS 52
SysAD7 51
SysAD8 50
VCC 49
VSS 48
SysAD9 47
Int0* 46
SysAD10 45
VCC 44
VSS 43
SysAD11 42
SysAD12 41
VCC 40
VSS 39
SysAD13 38
SysAD14 37
VCC 36
VSS 35
SysAD15 34
SysAD16 33
VCC 32
VSS 31
VSS 61
VCC 62
JTDI 63
SysAD4 64
JTDO 65
SysAD3 66
VSS 67
VCC 68
SysAD2 69
SysAD1 70
VSS 71
VCC 72
SysAD0 73
PReq* 74
VSS 75
VCC 76
SysAD31 77
PValid* 78
VSS 79
VCC 80
SysAD30 81
EOK* 82
SysAD29 83
VSS 84
VCC 85
SysAD28 86
SysAD27 87
Int2* 88
VSS 89
VCC 90
30 VSS
29 VCC
28 Int4*
27 SysAD17
26 SysAD18
25 VSS
24 SyncIn
23 VCC
22 SysAD19
21 SyncOut
20 VSS
19 VCC
18 TClock
17 VSS
16 MasterClock
15 VCC
14 VSSP
13 VCCP
12 PLLCap1
11 PLLCap0
10 VSSP
9 VCCP
8 VCC
7 SysAD20
6 VSS
5 VCC
4 SysAD21
3 SysAD22
2 VSS
1 VCC
91 VCC
92 VSS
93 NMI*
94 SysAD26
95 PMaster*
96 VCC
97 VSS
98 SysAD25
99 EReq*
100 SysCmd0
101 VCC
102 VSS
103 SysCmd1
104 Reset*
105 EValid*
106 SysCmd2
107 VCC
108 VSS
109 SysCmd3
110 ColdReset*
111 SysCmd4
112 DivMode1
113 VCC
114 VSS
115 SysAD24
116 DivMode0
117 SysAD23
118 Int3*
119 VCC
120 VSS

Addressing Modes

Introduction

MIPS has only a small number of ways that is computes addresses in memory. The address can be an address of an instruction (for branch and jump instructions) or it can be an address of data (for load and store instructions).

We'll look at the four ways addresses are computed

- **Register Addressing** This is used in the **jr** (jump register) instruction.
- **PC-Relative Addressing** This is used in the **beq** and **bne** (branch equal, branch not equal) instructions.
- **Pseudo-direct Addressing** This is used in the **j** (jump) instruction.
- **Base Addressing** This is used in the **lw** and **sw** (load word, store word) instructions.

We'll also consider *indirect addressing*, a popular addressing mode in CISC ISAs.

Register Addressing

Register addressing is used in the **jr** instruction. Because a register stores 32 bits, and because an address in a MIPS CPU is also 32 bits, you can specify any address in memory.

The typical call is:

```
  jr $rs
```

where **$rs** is replaced by any register.

The semantics of this is:

```
  PC <- R[s]
```

This means the PC (program counter) is updated with the contents of register **s**. Recall that a jump or branch is updated by modifying the contents of the program counter.

Register addressing gives you the ability to generate any address in memory. An *address exception* occurs if the two low bits are not 00.

PC-Relative Addressing

PC-relative addressing occurs in branch instructions, **beq** and **bne** (and other variations of branch instructions).

These instructions are I-type instructions, with the following format.

Opcode	Register s	Register t	Immediate
B$_{31-26}$	**B$_{25-21}$**	**B$_{20-16}$**	**B$_{15-0}$**
ooo ooo	**sssss**	**ttttt**	**iiii iiii iiii iiii**

PC <- PC + sign-ext$_{32}$(IR$_{15-0}$::00)

You take the 16 bit immediate value, add two zeroes to the end (which is the same as shifting it logical left 2 bits). This creates a value that's divisible by 4. Then, you sign-extend it to 32 bits, and add it to the PC.

Thus, the range of possible addresses is: **PC - 2^{17}** up to **PC + (2^{17} - 4)**.

2^{17} is 128 K. So you can jump back roughly -128,000 bytes backwards up to about 128,000 bytes forward. That's large, but still a small fraction of memory. Fortunately, for branch instructions, you don't need to jump that far, if you've written reasonably good code.

Pseudo-Direct Addressing

Direct addressing means specifying a complete 32 bit address in the instruction itself. However, since MIPS instructions are 32 bits, we can't do that. In theory, you only need 30 bits to specify the address of an instruction in memory. However, MIPS uses 6 bits for the opcode, so there's still not enough bits to do true direct addressing.

Instead, we can do pseudo-direct addressing. This occurs in **j** instructions.

Opcode	Target
B$_{31-26}$	**B$_{25-0}$**
ooo ooo	**tt tttt tttt tttt tttt tttt tttt**

26 bits are used for the target. This is how the address for pseudo-direct addressing is computed.

PC <- PC$_{31-28}$::IR$_{25-0}$::00

Take the top 4 bits of the PC, concatenate that with the 26 bits that make up the target, and concatenate that with 00. This produces a 32 bit address. This is the new address of the PC.

This allows you to jump to 1/16 of all possible MIPS addresses (since you don't control the top 4 bits of the PC).

Base Addressing

The other three addressing modes modify the PC. They create addresses for branch/jump instructions.

However, load/store instructions also generate addresses in memory.

Let's consider the following instruction.

```
  lw $rt, offset($rs)
```

where **$rs** and **$rt** are any two registers.

The offset is stored in 16 bits 2C. Thus, **lw** and **sw** are I-type instructions.

The address computed is:

$$\texttt{addr <- R[s] + sign-ext}_{32}\texttt{(offset)}$$

$rs is the base register, which is where the name base addressing comes from.

The **offset** is the 16 bit immediate value from the instruction. Unlike branch instructions, we don't add a 00 to the end of the immediate value, even though we can only load and store are word-aligned addresses.

The reason is because there are other load/store instructions that load/store halfwords and bytes, and it makes sense to compute the addresses the same way, regardless of what you're loading.

So what happens an address is computed that's not word aligned? An address exception occurs.

Indirect Addressing

MIPS does not support indirect addressing. RISC ISAs generally do not support such an instruction. However, it's a popular addressing mode for CISC ISAs.

Let's see how this indirect addressing works. First, recall how **lw** works. It adds the contents of a register to a sign-extended offset. This results in an address. The word stored at that address is loaded into a register.

Let's make up a new instruction called **indlw** which means "indirect load word". This instruction doesn't really exist in MIPS, but pretend it does.

```
  indlw $rt, offset($rs)
```

In this case, we'll do the same thing. Add the sign-extended offset to **$rs**. But instead of loading the word at that address into a register, we load the word (say, to some temporary location like $at), and use that word as an address, then we load that word from memory.

Thus, we go to memory twice. First time, we load a word that represents an address, and second time, we use that address to load a word of data.

The semantics are:

$$R[\ t\ ] \texttt{<-} M_4[\ M_4[\ R[\ s\ ] + \text{sign-ext}_{32}(\ \text{offset}\ )\ ]\ ]$$

Indirect addressing might seem odd, but if you increment the address stored in memory, it's one way to process an array. That address could be a pointer.

Instruction Set

Instruction name	Mnemonic	Format	Encoding			
Load Byte	LB	I	32_{10}	rs	rt	offset
Load Halfword	LH	I	33_{10}	rs	rt	offset
Load Word Left	LWL	I	34_{10}	rs	rt	offset
Load Word	LW	I	35_{10}	rs	rt	offset
Load Byte Unsigned	LBU	I	36_{10}	rs	rt	offset
Load Halfword Unsigned	LHU	I	37_{10}	rs	rt	offset
Load Word Right	LWR	I	38_{10}	rs	rt	offset
Store Byte	SB	I	40_{10}	rs	rt	offset

Store Halfword	SH	I	41_{10}	rs	rt	offset
Store Word Left	SWL	I	42_{10}	rs	rt	offset
Store Word	SW	I	43_{10}	rs	rt	offset
Store Word Right	SWR	I	46_{10}	rs	rt	offset

Instruction name	Mnemonic	Format	Encoding					
Add	ADD	R	0_{10}	rs	rt	rd	0_{10}	32_{10}
Add Unsigned	ADDU	R	0_{10}	rs	rt	rd	0_{10}	33_{10}
Subtract	SUB	R	0_{10}	rs	rt	rd	0_{10}	34_{10}
Subtract Unsigned	SUBU	R	0_{10}	rs	rt	rd	0_{10}	35_{10}
And	AND	R	0_{10}	rs	rt	rd	0_{10}	36_{10}
Or	OR	R	0_{10}	rs	rt	rd	0_{10}	37_{10}
Exclusive Or	XOR	R	0_{10}	rs	rt	rd	0_{10}	38_{10}
Nor	NOR	R	0_{10}	rs	rt	rd	0_{10}	39_{10}
Set on Less Than	SLT	R	0_{10}	rs	rt	rd	0_{10}	42_{10}
Set on Less Than Unsigned	SLTU	R	0_{10}	rs	rt	rd	0_{10}	43_{10}

Add Immediate	ADDI	I	8_{10}	rs	rd	immediate
Add Immediate Unsigned	ADDIU	I	9_{10}	\$s	\$d	immediate
Set on Less Than Immediate	SLTI	I	10_{10}	\$s	\$d	immediate
Set on Less Than Immediate Unsigned	SLTIU	I	11_{10}	\$s	\$d	immediate
And Immediate	ANDI	I	12_{10}	\$s	\$d	immediate
Or Immediate	ORI	I	13_{10}	\$s	\$d	immediate
Exclusive Or Immediate	XORI	I	14_{10}	\$s	\$d	immediate
Load Upper Immediate	LUI	I	15_{10}	0_{10}	\$d	immediate

Instruction name	Mnemonic	Format	Encoding					
Shift Left Logical	SLL	R	0_{10}	0_{10}	rt	rd	ra	0_{10}
Shift Right Logical	SRL	R	0_{10}	0_{10}	rt	rd	sa	2_{10}
Shift Right Arithmetic	SRA	R	0_{10}	0_{10}	rt	rd	sa	3_{10}
Shift Left Logical Variable	SLLV	R	0_{10}	rs	rt	rd	0_{10}	4_{10}
Shift Right Logical Variable	SRLV	R	0_{10}	rs	rt	rd	0_{10}	6_{10}
Shift Right Arithmetic Variable	SRAV	R	0_{10}	rs	rt	rd	0_{10}	7_{10}

Instruction name	Mnemonic	Format	Encoding					
Move from HI	MFHI	R	0_{10}	0_{10}	0_{10}	rd	0_{10}	16_{10}
Move to HI	MTHI	R	0_{10}	rs	0_{10}	0_{10}	0_{10}	17_{10}
Move from LO	MFLO	R	0_{10}	0_{10}	0_{10}	rd	0_{10}	18_{10}
Move to LO	MTLO	R	0_{10}	rs	0_{10}	0_{10}	0_{10}	19_{10}
Multiply	MULT	R	0_{10}	rs	rt	0_{10}	0_{10}	24_{10}
Multiply Unsigned	MULTU	R	0_{10}	rs	rt	0_{10}	0_{10}	25_{10}
Divide	DIV	R	0_{10}	rs	rt	0_{10}	0_{10}	26_{10}
Divide Unsigned	DIVU	R	0_{10}	rs	rt	0_{10}	0_{10}	27_{10}

Instruction name	Mnemonic	Format	Encoding					
Jump Register	JR	R	0_{10}	rs	0_{10}	0_{10}	0_{10}	8_{10}
Jump and Link Register	JALR	R	0_{10}	rs	0_{10}	rd	0_{10}	9_{10}
Branch on Less Than Zero	BLTZ	I	1_{10}	rs	0_{10}	offset		
Branch on Greater Than or Equal to Zero	BGEZ	I	1_{10}	rs	1_{10}	offset		

Branch on Less Than Zero and Link	BLTZAL	I	1_{10}	rs	16	offset
Branch on Greater Than or Equal to Zero and Link	BGEZAL	I	1_{10}	rs	17	offset
Jump	J	J	2_{10}	instr_index		
Jump and Link	JAL	J	3_{10}	instr_index		
Branch on Equal	BEQ	I	4_{10}	rs	rt	offset
Branch on Not Equal	BNE	I	5_{10}	rs	rt	offset
Branch on Less Than or Equal to Zero	BLEZ	I	6_{10}	rs	0_{10}	offset
Branch on Greater Than Zero	BGTZ	I	7_{10}	rs	0_{10}	offset