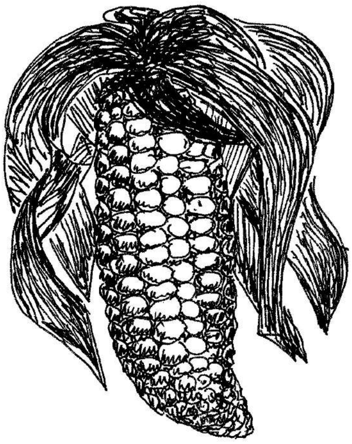


Manual
Mais Processor

a 32 bit processor written in VHDL

Dossmatik GmbH
Karlstraße 41
04420 Markranstädt
Germany

Autor:	René Doß
Date:	6.12.2013
Version:	1.0

Disclaimer, Warranty

Dossmatik GmbH is not responsible for the end product or success of the end product in any way. The final sign and acceptance test of any products or designs is the sole responsibility of the customer. The processor core is permanent under development and does not pass the full acceptance test. Many features are working and can be used. It can't garantee the whole preformance.

The manual is written to give you quick start and an overview of the core. Please do not hesitate to contact us. Any requests are welcome.

Contents

preamble

Sometimes it is laborious to solve all functions in a HDL description. Often some tasks are simplier and better to implement in a programming language like C or Assembler. The development in HDL takes more time und you will reduce your effort when you can write some parts in software. The border of hard- and software is a design decision. An implemention on a FPGA has a leak. There is no microcontroller available. The scope of the Mais processor is a linking element between some hardware and software descriptions. The core is written in VHDL and can simulate in a VHDL simulator like GHDL.

There are good documentation about MIPS architecture [2]. This is not the first open source implementation but this is the first practical impementation with a starting point for soft core beginners. The VHDL code is written without any spezial elements. This makes the code is portable to all FPGAs. Mais is a huge reimplementation. The endeavour was to get an optional processor.

System architechture is an effective 32-bit RISC processor with 5 pipe stages. One instruction can execute on one clock cycle. Only the load from memory takes longer.

A MIPS compiler can produce hexcode for the Mais processor. Also the GNU toolchain is useable for software coding.

I deliver also makefiles for some tools like compiler or simulation. Makefile is an excelent tool to enhance your productivity.

I hope you will choose Mais for your work.

Chapter 1

Introduction

FPGA has only logical gates and simple digital elements. A hardware description language is used to build lager blocks with higher functionality. A special function is a processor. It can execute software and divide the design in Hardware an Software development. This is an universal part. A processor is well familiar in emdedded design. A softcore give more flexibility the product.

Mais-CPU is written in VHDL. Havard architecture separates Instruction RAM und Data RAM in separate memories. MAIS-CPU is compatible to MIPS instruction Set Architecture. All instructions are 32bit width. It is possible to use a GNU C-Compiler. Calculating of data and statemachines can be written in software. A bus interface connects devices with the CPU. This implementation is SoC (system on Chip).

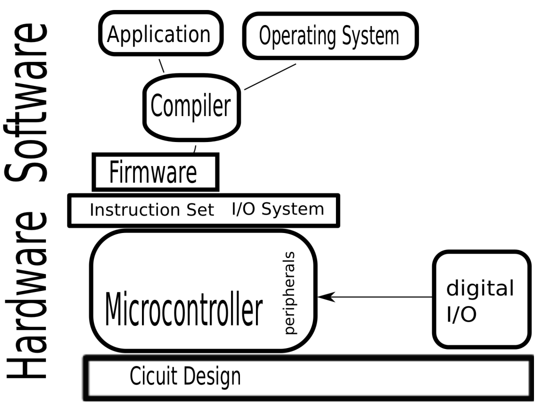


Figure 1.1: embedded design

The CPU is designed in a typical pipeline architecture. The advantage is, in each clock cycle is equal one maschine cycle. At a branch the followed instruction is also executed. It is the branch delayed slot. Mais was developed as single core. A single core needs no data synchronisation. A good general book about MIPS is [4].

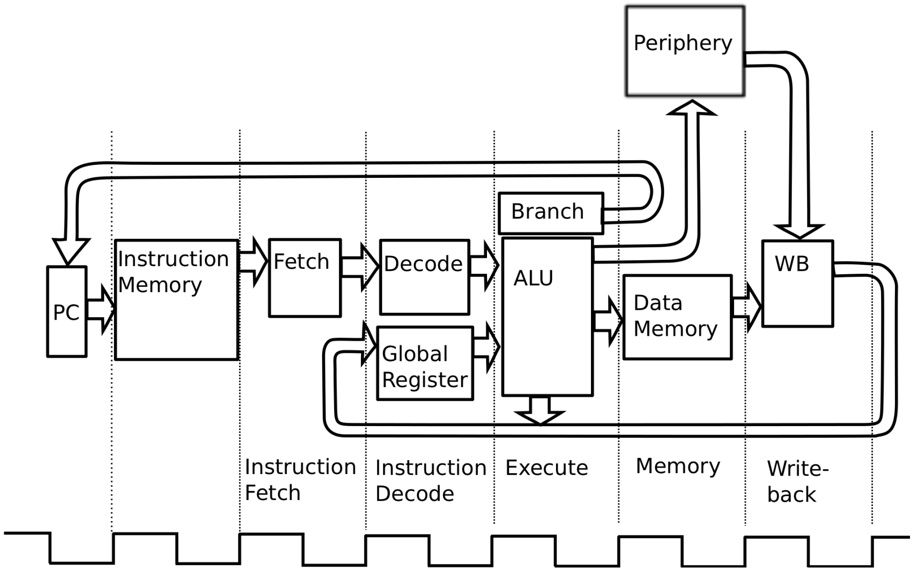


Figure 1.2: Mais architecture

Chapter 2

Pipeline

2.1 pipelining basics

An unpiplined system has a reduced throughout [4]. The combinatory logic is strong branched. One instruction must complete before next instruction can begin. The clockrate is lower or one instruction need several clock periodes. Several instructions are processed in a pipelining architecture at the same time. The treatment is not parallel, each instruction is in another state. In each state a particular task is carrying out. Followed instructions are placed sequentially into the pipe.

The store instructions are straight away. Register values are written in memory. No dataflow is backward. The register values in the pipline are valid for all cases. All followed instructions are in indepentent in pipeline. No conflicts are possible.

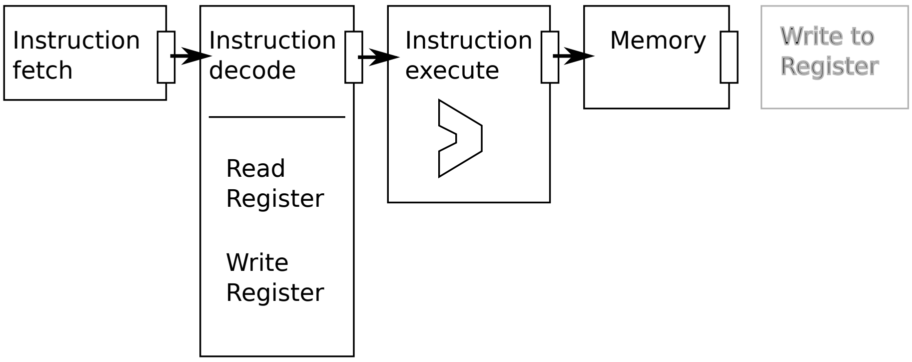


Figure 2.1: store pipe

The load instruction takes values from memory into the Register.

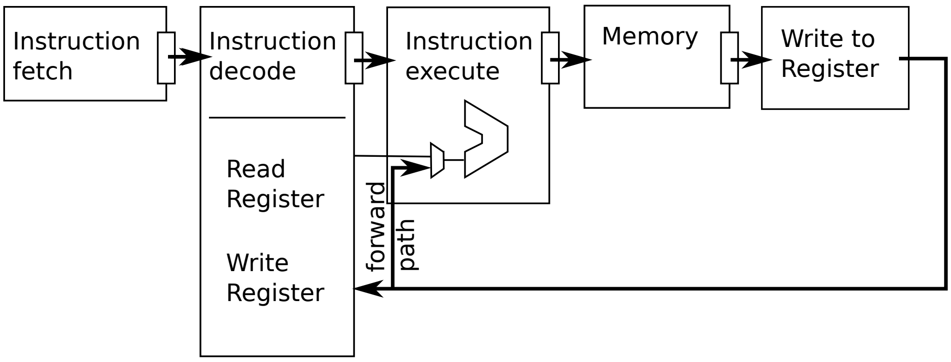


Figure 2.2: load operation

Alu instruction uses values form register and writes data in the registers back. A critical situation accurs when a result is stored in a register and the next instruction use this result. Normaly a pipelining hazard exists. The Mais processor has a forwarding path. Extra Hardware resolves this hazard. The pre-existing value is catched in the pipeline and is overwrite in the execute state.

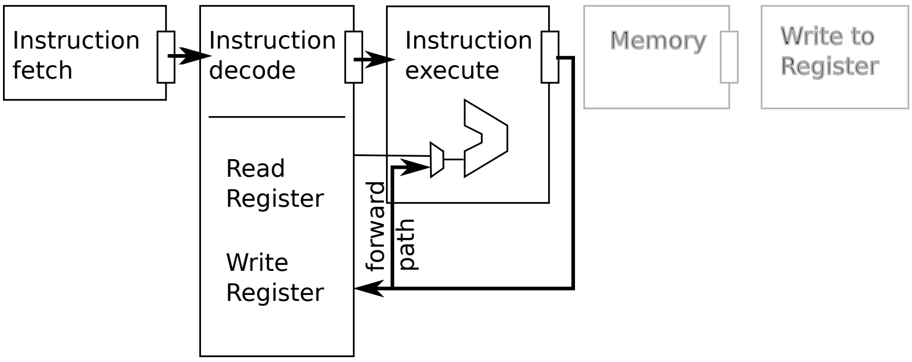


Figure 2.3: alu operation

Chapter 3

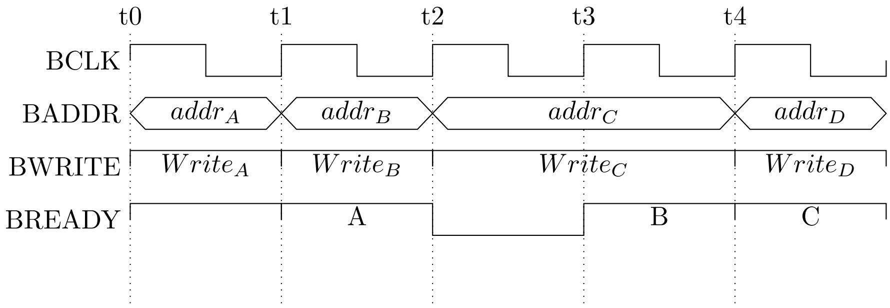
peripherals

3.1 Bus Introduction

The CPU communicates over a bus to all peripheral devices. A bus is a typical interface in data extensions system. The bus is the simple peripheral bus. This is single slave bus architecture. CPU is master and all other devices interact as slaves.

All signals are active HIGH.

Name	**Source**	**Description**
BCLK	global	this is the CPU clock and the bus runs with this clock
BRESET	global	high signal reset the slave
BADDR[31:0]	master	address
HSEL [x:0]	master	slave select signals matchs too the highest address bits
BWR	master	indicate a read transfer
BRD	master	indicate a write transfer
BACTIVE	master	indicate a write or read transfer
BMASK [3:0]	master	indicate ative bytes in transfer
BWDATA[31:0]	master	write data during write transfer from master to slave
BREADY	master	signals to slave are valid
BRDATA[31:0]	slave	read data during read transfer from slave to master
BREADYOUT	slave	when High the transfer complete, when Low interlock the bus



3.2 multi slave bus

The Mais CPU is wired in MAIS_soc.vhd. The Memory is in the example bus slave 0 at address 0x00000000. Slave 1 is an UART for simple communication at address 0x20000000. The third slave is a bus dummy. A bus dummy puts out definded and valid signal on the bus, when no data trafic is on the bus.

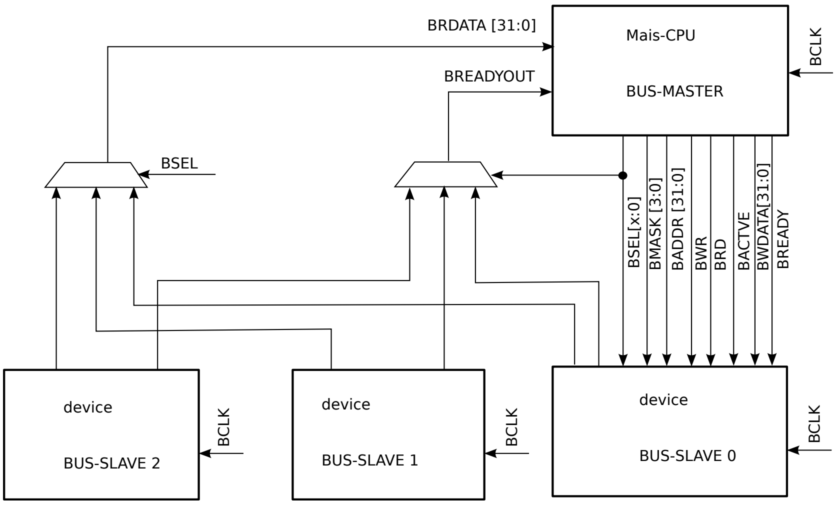


Figure 3.1: multi slave bus

Chapter 4

Crosscompile

```
############################################################
# makefile for build mips tool chain
#
# written by René Doss
#
# after preliminary work by
# Dmitriy Schapotschkin and Martin Strubel
############################################################



BINUTILS = binutils-2.22
GCC = gcc-4.4.3
NEWLIB = newlib-1.20.0
GDB = gdb-7.5
:

ARCHITECTURE = mips-elf
PREFIX =/opt/mips


GCC_OPTS =  \
--with-newlib\
--with-float=soft\
--disable-nls \
--disable-threads \
--disable-shared \
--disable-libssp \
--with-gnu-ld --with-gnu-as


.PHONY:  download install-newlib install-binutils install-gcc

download:
```

```
#download binutils
wget ftp.gnu.org/gnu/binutils/$(BINUTILS).tar.gz
tar -xzf $(BINUTILS).tar.gz
#download gcc
wget ftp://ftp.gwdg.de/pub/misc/gcc/releases/$(GCC)/$(GCC).tar.gz
tar -xzf $(GCC).tar.gz
#download newlib
wget ftp://sources.redhat.com/pub/newlib/newlib-1.20.0.tar.gz
tar -xzf newlib-1.20.0.tar.gz


download-gdb:
#download gdb
wget ftp://ftp.gnu.org/gnu/gdb/$(GDB).tar.gz
tar -xzf $(GDB).tar.gz

build-gdb:
mkdir gdb
cd gdb; \
../$(GDB)/configure --target=$(ARCHITECTURE) --prefix=$(PREFIX) --without-auto-load-safe-path
make all 2>&1 | tee gdb_build.log

install-gdb:
cd gdb;\
make install 2>&1 | tee gdb_install.log

build-binutils:
mkdir binutils
cd binutils; \
../$(BINUTILS)/configure --prefix=$(PREFIX) --target=$(ARCHITECTURE) 2>&1 | tee binutils_conf
make all 2>&1 | tee binutils_make.log

install-binutils:
cd binutils; \
make install 2>&1 | tee binutils_install.log


build-gcc-bs:
mkdir gcc-bootstrap
export PATH=$$PATH:$(PREFIX)/bin; \
cd gcc-bootstrap; \
../$(GCC)/configure --target=$(ARCHITECTURE) \
$(GCC_OPTS) --without-headers \
--prefix=$(PREFIX)  2>&1 |tee gcc-bs_configure.log;\
make all-gcc 2>&1 | tee gc-bs_make.log

install-gcc-bs:
cd gcc-bootstrap; \
make install-gcc 2>&1 | tee gcc-bs_install.log
```

```
build-newlib:
mkdir newlib
cd newlib \
export PATH=$$PATH:$(PREFIX)/bin; \
../$(NEWLIB)/configure --target=$(ARCHITECTURE) --prefix=$(PREFIX) --with-float=soft ; \
make all 2>&1 | tee newlib.log
```

```
#hier weiter aufraumen

install: install-binutils install-gdb install-gcc


install-newlib:
export PATH=$$PATH:$(PREFIX)/bin; \
$(MAKE) -C newlib install
```

```
.PHONY: download build-newlib build-binutils build-mpfr build-gcc
```

```
build: build-binutils build-gcc


newlib/config.status: $(NEWLIB)/configure
[ -e newlib ] || mkdir newlib
export PATH=$$PATH:$(PREFI)/bin; \
cd newlib; \
$(NEWLIB)/configure --target=$(ARCHITECTURE) \
--prefix=$(PREFIX)
```

```
gcc-newlib/config.status: $(UNISRC)/gcc/configure
[ -e gcc-newlib ] || mkdir gcc-newlib
cd gcc-newlib; \
../$(UNISRC)/configure --target=$(ARCHITECTURE) \
$(GCC_OPTS) --with-newlib \
--prefix=$(INSTALL_PREFIX) \
--with-sysroot=$(INSTALL_PREFIX) \
--with-build-sysroot=$(BUILD_PREFIX)
```

Chapter 5

Assembly Language

There are 31 general purpose 32bit registers. Register 0 has a special function. It can be written but the readout value is allways constant 0. The register 31 has also a special application in some branch function the return adress is saved in branch link instruction. The other registers are without special effects. Typical convention applies by compiler.

```
#include <mips/asm.h>
#include <mips/regdef.h>
```

Hardware Name	Common Name	Description
$0	zero	zero register always has the value 0
$1	at	Assembler temporary
$2-$3	v0-v1	function result register
$4-$7	a0-a3	function argument
$8-$15	t0-t7	temporary, saved by caller
$16-$23	s0-s7	temporary
$24-$25	t8-t9	temporary
$26-$27	k0-k1	reseved for OS
$28	gp	global data pointer to data segment
$29	sp	stack pointer
$30	fp or s8	frame pointer
$31	ra	return address

There are three special registers. The PC (program counter) holds the address of the next instruction. Only implicity access modifies by certain instructions.
If the design has a multiplyer, two additional registers are available HI and LOW registers. These are the result registers of multiplying and division operation. Multiply instruction has a result of 64bit. Divide instruction placing the quotient and a reminder in HI an LOW register. Integer multiplies and divide calculation need some more clock cycles. It runs parallel with other instructions.

5.0.1 pointer operation

```
sw   $26 , GDBState
la   $26 , (input_buffer)
```

5.0.2 GNU inline asm

Sometimes it is nessarry to combine C and simple ams. The inline assembly have to write in double quotes. The gcc sends the instruction as string to as. The basic format of inline assembly is

```
asm("assembly code");
```

If more than one instruction are written, it have to separated by `"\n\t"` .

```
asm("assembly code \n\t"
    "assembly code"
    );
```

The general form with operands is

```
asm(" asm code ": output operand list : input operand list);
asm ( "mfc0   %[result] ,$12":[result] "=r"(var));
  asm ( "mtc0   %[value] ,$14": :[value] "r"(*ptr));
```

Chapter 6

Instructions

6.1 Overview Instruction

[4] [1] [3]

6.1.1 arithmetic instructions

Mnemonic	Description
ADD	add
ADDI	add immediate word
ADDIU	add immediate unsigned word
ADDU	add immediate unsigned word
SUB	subtract word
SUBU	subtract unsigned word
DIV	divide word**
DIVU	divide unsigned word**
MULT	multiply word**
MULTU	multiply unsigned word**
MFHI	move from HI register
MFLO	move from LO register
MTHI	move to HI register
MTLO	move to LO register

6.1.2 memory instructions

Datas can be only moved between memory and the CPU general registers by load and store instructions. For different data lengths exist are also the correct load and store.

Mnemonic	Description
LB	load byte
LBU	load byte unsigned
LH	load halfword
LHU	load halfword unsigned
LW	load word
LWL	load word left
LWR	load word right
LWU	load word unsigned
SB	store byte
SH	store halfword
SW	store word
SWL	store word left
SWR	store word right

The instuction calculates the access address from the content of one register and an fixed offset. The offset can be negative.

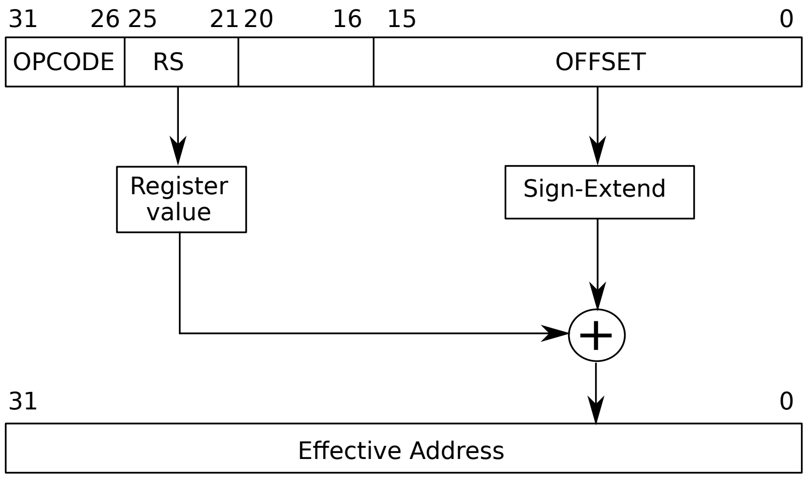


Figure 6.1: Address calculation for Loads and Stores

The specific mnemonic is such as

```
lw t0,20(t1)
```

A pseudo instruction exist in assembler.

```
lw t0,0x12345678
```

and expand to

```
lui at,0x1234
lw t0,0x5678(at)
```

6.1.3 logic instructions

Mnemonic	Description
AND	And
ANDI	And Immediate
LUI	load upper immediate
NOR	not or
OR	or
ORI	or immediate
SLL	shift word left logical
SLLV	shift word left logical variable
SLT	set on less than
SLTI	set on less than immediate
SLTIU	set on less than immediate unsigned
SLTU	set on less than unsigned
SRA	shift word right arithmetic
SRAV	shift word right arithmetic variable
SRL	shift word right logical
SRLV	shift word right logical variable

6.1.4 special instructions

Mnemonic	Description
BREAK	Breakpoint
SYSCALL	syscall
ERET	return from exception
NOP	no operation

6.1.5 jump and branch instructions

Mnemonic	Description
B	Branch**
BAL	Branch and link**
BEQ	Branch on equal
~~BEQL~~	Branch on equal likely*
BEQZ	Branch on equal zero**
BGE	Branch on greater than equal**
BGEU	Branch on greather than equal unsigned**
BGEZ	Branch on greater than or equal to zero
BGETAL	Branch on greater than or equal to zero and link
~~BGEZALL~~	Branch on greater than or equal to zero and link likely*
~~BGEZL~~	Branch on greater than or equal to zero likely*
BGT	Branch on greater than**
BQTU	Branch on greater then unsigned**
BGTZ	Branch on greater than zero
~~BGTZL~~	Branch on greater than zero likely*
BLE	Branch on less than equal**
BLEU	Branch on less than equal unsigned**
BLEZ	branch on less than or equal to zero
~~BLEZL~~	Branch on less than or equal to zero likely*
BLT	Branch on less than**
BLTU	Branch on less than unsigned**
BLTZ	Branch on less than zero
BLTZAL	Branch on less than zero and link
~~BLTZALL~~	Branch on less zero and link likely*
BLTZL	Branch on less than zero likely*
BNE	Branch on not equal
~~BNEL~~	Branch on not equal likely*
BNEZ	Branch on not equal zero**
J	Jump
JAL	Jump and link
JALR	Jump and link register
JR	jump register

* likely is not implemented, use gcc option -mno-branch-likely

** pseudoinstruction

A delay slot may not itself occupied by a jump or branch instruction. This combination of opcodes has an unpredictable situation. Also a had crash is possible that only a reset can resolve it.

6.2 Instructions

ADD

31	25	20	15	10	5
OP 000000	rs	rt	rd	0 00000	ADD 100000
6	5	5	5	5	6

Format: ADD rd,rs,rt

Purpose: to add two register in 32 bit integer format
Operation: $\boldsymbol{r[rd] \leftarrow r[rs] + r[rt]}$

ADDI add immediate

31	25	20	15
OP 001000	rs	rt	immediate
6	5	5	16

Format: ADDI rd,rs,immediate

Purpose: to add a constant to a register
Operation: $\boldsymbol{r[rt] \leftarrow r[rs] + immediate}$

ADDIU add immediate unsigned

31	25	20	15
OP 001001	rs	rt	immediate
6	5	5	16

Format: ADDIU rd,rs,immediate

Purpose: to add a constant to a register
Operation: $\boldsymbol{r[rt] \leftarrow r[rs] + immediate}$

ADDU

31	25	20	15	10	5
OP 000000	rs	rt	rd	0 00000	ADD 100001
6	5	5	5	5	6

Format: ADDU rd,rs,rt

Purpose: to add two register in 32 bit integer format
Operation: $\boldsymbol{r[rd] \leftarrow r[rs] + r[rt]}$

AND

31	25	20	15	10	5
OP 000000	rs	rt	rd	0 00000	ADD 100100
6	5	5	5	5	6

Format: AND rd,rs,rt

Purpose: two register bitwise logical AND
Operation: $\boldsymbol{r[rd] \leftarrow r[rs]\ and\ r[rt]}$

ANDI and immediate

31	25	20	15
OP 001100	rs	rt	immediate
6	5	5	16

Format: ANDI rd,rs,immediate

Purpose: bitwise logical AND with a constant
Operation: $\boldsymbol{r[rt] \leftarrow r[rs]\ and\ immediate}$

BEQ Branch on equal

31	25	20	15
OP 000100	rs	rt	offset
6	5	5	16

Format: BEQ rd,rs,offset

Purpose: branch on equal relative offset
Operation: $\boldsymbol{if\ r[rs] = r[rt]\ then\ pc\ \leftarrow\ pc + (offset << 2)}$

BGEZ Branch on greater than or equal to zero

31	25	20	15
OP 000001	rs	00001	offset
6	5	5	16

Format: BGEZ rs,offset

Purpose: branch on equal relative offset;
Operation: $\boldsymbol{if\ r[rs] \geq 0\ then\ pc\ \leftarrow\ pc + (offset << 2)}$

BGEZAL Branch on greater than or equal to zero and link

31	25	20	15
OP 000001	rs	10001	offset
6	5	5	16

Format: BGEZAL rs,offset

Purpose: branch on greater than or equal relative offset; rescue the link address in R[31]
Operation: $\boldsymbol{if\ r[rs] \geq 0\ then\ pc\ \leftarrow\ pc + (offset << 2)}$
$\boldsymbol{r[31]\ \leftarrow\ pc + 8}$

BGTZ Branch on greater than zero

31	25	20	15
OP 000111	rs	00000	offset
6	5	5	16

Format: BGTZ rs,offset

Purpose: branch on greater than zero relative offset
and clear the delay slot if no branch is taken
Operation: $\boldsymbol{if\ r[rs] > 0\ then\ pc\ \leftarrow\ pc + (offset << 2)}$

BLEZ Branch on less than or equal to zero

31	25	20	15
OP 000111	rs	00000	offset
6	5	5	16

Format: BLEZ rs,offset

Purpose: branch on on less than or equal to zero relative offset
Operation: $\boldsymbol{if\ r[rs] \leq 0\ then\ pc\ \leftarrow\ pc + (offset << 2)}$

BLEZL Branch on less than or equal to zero likely

31	25	20	15
OP 010110	rs	00000	offset
6	5	5	16

Format: BLEZL rs,offset

Purpose: branch on on less than zero relative offset
Operation: $\boldsymbol{if\ r[rs] \leq 0\ then\ pc\ \leftarrow\ pc + (offset << 2)}$

Likely is not implemented. Behaviour is the same like instruction BLEZ.

BLTZ Branch on less than zero

31	25	20	15
OP 000001	rs	00000	offset
6	5	5	16

Format: BLTZ rs,offset

Purpose: branch on on less than zero relative offset
Operation: $\boldsymbol{if\ r[rs] < 0\ then\ pc\ \leftarrow\ pc + (offset << 2)}$

BLTZAL Branch on less than zero and link

31	25	20	15
OP 000001	rs	10000	offset
6	5	5	16

Format: BLTZAL rs,offset

Purpose: branch on less than zero relative offset; rescue the link address in R[31]
Operation: $\boldsymbol{if\ r[rs] < 0\ then\ pc\ \leftarrow\ pc + (offset << 2)}$
$\boldsymbol{r[31]\ \leftarrow\ pc + 8}$

BLTZL Branch on less than zero likely

31	25	20	15
OP 000001	rs	00010	offset
6	5	5	16

Format: BLTZ rs,offset

Purpose: branch on less than zero relative offset
Operation: $\boldsymbol{if\ r[rs] < 0\ then\ pc\ \leftarrow\ pc + (offset << 2)}$

Likely is not implemented. Behaviour is the same like instruction BLTZ.

BNE Branch on not equal

31	25	20	15
OP 000101	rs	rt	offset
6	5	5	16

Format: BNE rs,rt,offset

Purpose: branch on not equal relative offset
Operation: $\boldsymbol{if\ r[rs] \neq r[rt]\ then\ pc\ \leftarrow\ pc + (offset << 2)}$

BREAK Breakpoint

31	25	15	5
OP 000000	code		001101
6	20	6	

Format: BREAK

Format: BREAK code

Purpose: stop and hold on
Operation: wait

Reserved Break codes:
BREAK 1 stop and generate an interrupt used in debugger
BREAK 7 ignored, generated in C-Code after multiply operation

DIV divide word

31	25	20	15	5
OP 000000	rs	rt	0 00 0000 0000	DIV 011010
6	5	5	10	6

Format: DIV rs,rt

Purpose: to devide two register in 32 bit integer signed format
Operation: $\boldsymbol{r[LO]R[HI] \leftarrow r[rs]/r[rt]}$

DIVU divide unsigned word

31	25	20	15	5
OP 000000	rs	rt	0 00 0000 0000	DIVU 011011
6	5	5	10	6

Format: DIVU rs,rt

Purpose: to devide two register in 32 bit integer unsigned format
Operation: $\boldsymbol{r[LO]R[HI] \leftarrow r[rs]/r[rt]}$

ERET return from exception

31	25	20	5
CP0 010000	10000	0 000000000000000	ERET 011000
6	5	15	6

Format: eret

Purpose: return from interupt service routine
Operation: $\boldsymbol{pc \leftarrow cp0(r[14])}$

The EPC (exeption restart address register) hold the rejump point. This register is located in CP0. The older instruction RFE is now obsolete. See also [4].

J Jump

31	25
OP 000010	index
6	26

Format: J label

Purpose: Simple jump within a $\mathbf{2^{28}}$ byte page. The upper PC bits are untouched. This instruction change PC.
Operation: $\boldsymbol{pc \leftarrow pc(bit31...bit28)|index|00}$

JAL Jump and Link

31	25
OP 000011	index
6	26

Format: JAL label

Purpose: Simple jump within a $\mathbf{2^{28}}$ byte page. The upper PC bits are untouched. This instruction change PC. Save PC in register $31.
Operation: $\boldsymbol{r[31] \leftarrow pc; pc \leftarrow pc(bit31...bit28)|index|00}$

JALR jump and link register

31	25	20	15	10	5
OP 000000	rs	00000	rd	0 00000	JALR 001001
6	5	5	5	5	6

Format: JALR rs rd=31
Format: JALR rd,rs

Purpose: jump to an address The new address value is in register rs and the current pc is saved in register rd. Generally is used register $31 for rd.
Operation: $\boldsymbol{r[rd] \leftarrow pc; pc \leftarrow r[rd]}$

JR jump register

31	25	20	5
OP 000000	rs	0 000000000000000	JR 001000
6	5	15	6

Format: JR rs

Purpose: jump to an address
Operation: $\boldsymbol{pc \leftarrow r[rs]}$

LA Load address

LA is a pseudo instruction. This instruction is often in asm code but this is only assembler macro. LA has different options. Se also [1] and [4].

Format:	macro instruction
la $2, 4($2)	addiu $2, $2, 4
la $2, 32bit	lui $2 bit 31...16 ori $2 bit 15...0
la $2, 32bit ($3)	lui $2 bit 31...16 ori $2 bit 15...0 addu $2, $2, $3

Examples:
la $25, main
la $20, 0x12345678
Purpose: load values into register

LB load byte

31	25	20	15
OP 100000	rs	rt	offset
6	5	5	16

Format: LB rt,offset(rs)

Purpose: load signed byte from memory and converted to signed word
Operation: $\boldsymbol{r[rt] \leftarrow\ memory(r[rs] + offset)}$
(upper bits are signed extended)

LBU load byte unsigned

31	25	20	15
OP 100100	rs	rt	offset
6	5	5	16

Format: LBU rt,offset(rs)

Purpose: load a byte from memory
Operation: $\boldsymbol{r[rt] \leftarrow \; memory(r[rs] + offset)}$

LH load halfword

31	25	20	15
OP 100001	rs	rt	offset
6	5	5	16

Format: LH rt,offset(rs)

Purpose: load signed half word from memory and converted to signed word
Operation: $\boldsymbol{r[rt] \leftarrow \; memory(r[rs] + offset)}$
(upper bits are signed extended)

LHU load halfword unsigned

31	25	20	15
OP 100101	rs	rt	offset
6	5	5	16

Format: LHU rt,offset(rs)

Purpose: load unsigned half word from memory and converted to unsigned word
Operation: $\boldsymbol{r[rt] \leftarrow \; memory(r[rs] + offset)}$

LUI load upper immediate

31	25	20	15
OP 001111	00000	rt	immediate
6	5	5	16

Format: LUI rt,immediate

Purpose: load a constant into higher two bytes
Operation: $\boldsymbol{r[rt] \leftarrow immediate << 16}$

LW load word

31	25	20	15
OP 100011	rs	rt	offset
6	5	5	16

Format: LW rt,offset(rs)

Purpose: load a word from memory
Operation: $\boldsymbol{r[rt] \leftarrow memory(r[rs] + offset)}$

LWL load word left

31	25	20	15
OP 100010	rs	rt	offset
6	5	5	16

Format: LWL rt,offset(rs)

Purpose: load half word from memory into upper two bits
Operation: $\boldsymbol{r[rt] \leftarrow memory(r[rs] + offset) << 16}$

LWR load word right

31	25	20	15
OP 100110	rs	rt	offset
6	5	5	16

Format: LWR rt,offset(rs)

Purpose: load half word from memory into lower two bits
Operation: $\boldsymbol{r[rt] \leftarrow\ memory(r[rs] + offset)\ \&\ 0x00FF}$

MFHI move from HI register

31	25	15	10	5
OP 000000	0000000000	rd	0 00000	MFHI 010000
6	10	5	5	6

Format: MFHI rd

Purpose: move HI register into general register
Operation: $\boldsymbol{r[rd] \leftarrow r[HI]}$

MFLO move from LO register

31	25	15	10	5
OP 000000	0000000000	rd	0 00000	MFLO 010010
6	10	5	5	6

Format: MFLO rd

Purpose: move LO register into general register
Operation: $\boldsymbol{r[rd] \leftarrow r[LO]}$

MTHI move to HI Register

31	25	20	5
OP 000000	rs	0 000000000000000	MTHI 001001
6	5	15	6

Format: MTHI rs

Purpose: general register into HI register
Operation: $\boldsymbol{r[HI]} \leftarrow \boldsymbol{r[rs]}$

MTLO move to LO Register

31	25	20	5
OP 000000	rs	0 000000000000000	MTLO 001011
6	5	15	6

Format: MTLO rs

Purpose: general register into LO register
Operation: $\boldsymbol{r[LO]} \leftarrow \boldsymbol{r[rs]}$

MULT multiply word

31	25	20	15	5
OP 000000	rs	rt	0 00 0000 0000	MULT 011000
6	5	5	10	6

Format: MULT rs,rt

Purpose: to multiply two register in 32 bit integer signed format
Operation: $\boldsymbol{r[LO]R[HI]} \leftarrow \boldsymbol{r[rs] * r[rt]}$

MULTU multiply unsigned word

31	25	20	15	5
OP 000000	rs	rt	0 00 0000 0000	MULTU 011001
6	5	5	10	6

Format: MULTU rs,rt

Purpose: to multiply two register in 32 bit integer unsigned format
Operation: $\boldsymbol{r[LO]R[HI] \leftarrow r[rs] * r[rt]}$

NOR not or

31	25	20	15	10	5
OP 000000	rs	rt	rd	0 00000	NOR 100111
6	5	5	5	5	6

Format: NOR rd,rs,rt

Purpose: two register logical not or
Operation: $\boldsymbol{r[rd] \leftarrow r[rs]\ nor\ r[rt]}$

NOP no operation

31	25	20	15	10	5
OP 000000	0 00000	00000	00000	0 00000	SLL 000000
6	5	5	5	5	6

Format: NOP

Purpose: do nothing but pc-counter is running, more gap filler, go to the next instruction
Operation: empty cycle

This is a pseudo instruction SLL 0,0,0.

OR or

31	25	20	15	10	5
OP 000000	rs	rt	rd	0 00000	OR 100101
6	5	5	5	5	6

Format: OR rd,rs,rt

Purpose: two register logical or
Operation: $\boldsymbol{r[rt] \leftarrow r[rs]\ or\ r[rt]}$

ORI or immediate

31	25	20	15
OP 001101	rs	rt	immediate
6	5	5	16

Format: ORI rt,rs,immediate

Purpose: register logical or immediate
Operation: $\boldsymbol{r[rd] \leftarrow r[rs]\ or\ immediate}$

SB store byte

31	25	20	15
OP 102000	rs	rt	offset
6	5	5	16

Format: SB rt,offset(rs)

Purpose: store byte into memory
Operation: $\boldsymbol{memory(r[rs] + offset)\ \leftarrow\ r[rt]}$

SH store halfword

31	25	20	15
OP 101001	rs	rt	offset
6	5	5	16

Format: SH rt,offset(rs)

Purpose: store half word into memory
Operation: $\boldsymbol{memory(r[rs] + offset)} \leftarrow \boldsymbol{r[rt]}$

SLL shift word left logical

31	25	20	15	10	5
OP 000000	0 00000	rt	rd	0 sa	SLL 000000
6	5	5	5	5	6

Format: SLL rd,rt,sa

Purpose: shift left constant position
Operation: $\boldsymbol{r[rd]} \leftarrow \boldsymbol{r[rt]} << \boldsymbol{sa}$

SLLV shift word left logical

31	25	20	15	10	5
OP 000000	0 rs	rt	rd	0 00000	SLLV 000100
6	5	5	5	5	6

Format: SLLV rd,rt,rs

Purpose: shift left number of bit by content register rs
Operation: $\boldsymbol{r[rd]} \leftarrow \boldsymbol{r[rt]} << \boldsymbol{r[rs]}$

SLT set on less than

31	25	20	15	10	5
OP 000000	rs	rt	rd	0 00000	SLT 101010
6	5	5	5	5	6

Format: SLT rd,rs,rt

Purpose: compare two registers of less than
Operation: $\boldsymbol{if\ r[rs] < r[rt]\ then}$
$\boldsymbol{r[rd] \leftarrow\ 1}$
$\boldsymbol{else}$
$\boldsymbol{r[rd] \leftarrow\ 0}$

SLTI set on less than immediate

31	25	20	15
OP 001010	rs	rt	immediate
6	5	5	16

Format: SLTI rd,rs,immediate

Purpose: compare register with immediate of less than
Operation: $\boldsymbol{if\ r[rs] < immediate\ then}$
$\boldsymbol{r[rt] \leftarrow\ 1}$
$\boldsymbol{else}$
$\boldsymbol{r[rt] \leftarrow\ 0}$

SLTIU set on less than immediate unsigned

31	25	20	15
OP 001011	rs	rt	immediate
6	5	5	16

Format: SLTIU rd,rs,immediate

Purpose: compare unsigned register with immediate of less than
Operation: $\boldsymbol{if\ r[rs] < immediate\ then}$
$\boldsymbol{r[rt] \leftarrow 1}$
$\boldsymbol{else}$
$\boldsymbol{r[rt] \leftarrow 0}$

SLTU set on less than unsigned

31	25	20	15	10	5
OP 000000	rs	rt	rd	0 00000	SLTU 101011
6	5	5	5	5	6

Format: SLTU rd,rs,rt

Purpose: compare unsigned two registers of less than
Operation: $\boldsymbol{if\ r[rs] < r[rt]\ then}$
$\boldsymbol{r[rd] \leftarrow 1}$
$\boldsymbol{else}$
$\boldsymbol{r[rd] \leftarrow 0}$

SRA shift word right arithmetic

31	25	20	15	10	5
OP 000000	0 00000	rt	rd	sa	SRA 000011
6	5	5	5	5	6

Format: SRA rd,rt,sa

Purpose: shift right constant position
Operation: $\boldsymbol{r[rd] \leftarrow r[rt] >> sa}$

SRAV shift word right arithmetic variable

31	25	20	15	10	5
OP 000000	0 rs	rt	rd	0 00000	SLLV 000111
6	5	5	5	5	6

Format: SRAV rd,rt,rs

Purpose: shift right number of bit by content register rs
Operation: $\boldsymbol{r[rd] \leftarrow r[rt] >> r[rs]}$

SRL shift word right logical

31	25	20	15	10	5
OP 000000	0 00000	rt	rd	0 sa	SRL 000010
6	5	5	5	5	6

Format: SRL rd,rs,sa

Purpose: shift right constant position
Operation: $\boldsymbol{r[rd] \leftarrow r[rt] >> sa}$

SRLV shift word right logical variable

31	25	20	15	10	5
OP 000000	0 rs	rt	rd	0 00000	SRLV 000110
6	5	5	5	5	6

Format: SRLV rd,rt,rs

Purpose: shift right number of bit by content register rs
Operation: $\boldsymbol{r[rd] \leftarrow r[rt] >> r[rs]}$

SUB subtract word

31	25	20	15	10	5
OP 000000	rs	rt	rd	0 00000	SUB 100010
6	5	5	5	5	6

Format: sub rd,rs,rt

Purpose: to subtract two register in 32 bit integer format
Operation: $\boldsymbol{r[rd] \leftarrow r[rs] - r[rt]}$

SUBU substract unsigned word

31	25	20	15	10	5
OP 000000	rs	rt	rd	0 00000	SUBU 100011
6	5	5	5	5	6

Format: SUBU rd,rs,rt

Purpose: to subtract two register in 32 bit integer format
Operation: $\boldsymbol{r[rd] \leftarrow r[rs] - r[rt]}$

SW store word

31	25	20	15
OP 101011	rs	rt	offset
6	5	5	16

Format: SW rt,offset(rs)

Purpose: store a word into memory
Operation: $\boldsymbol{memory(r[rs] + offset) \ \leftarrow \ r[rt]}$

SWL store word left

31	25	20	15
OP 101010	rs	rt	offset
6	5	5	16

Format: SWL rt,offset(rs)

Purpose: store the most significant part of a word to an unaligned memory address
Operation: $\boldsymbol{memory(r[rs] + offset) \leftarrow r[rt]}$

Memory contents

i	j	k	l

Memory contents				$\boldsymbol{Addr_{1..0}}$
A	B	C	D	0
i	B	C	D	1
i	j	C	D	2
i	j	k	D	3

SWR store word right

31	25	20	15
OP 101110	rs	rt	offset
6	5	5	16

Format: SWR rt,offset(rs)

noch mal genau die Implementierung überprüfen Purpose: store a word into memory
Operation: $\boldsymbol{memory(r[rs] + offset) \leftarrow r[rt]}$

Memory contents

i	j	k	l

Memory contents				$\boldsymbol{Addr_{1..0}}$
A	j	k	l	0
A	B	k	l	1
A	B	C	l	2
A	B	C	D	3

Bibliography

[1] Ervin Farquhar and Philip Bunce. The MIPS Programmer´s Handbook. Morgan Kaufmann. 1994.

[2] David A. Patterson and John L Hennessy. 4. Auflage. Oldenburg Verlag, 2009.

[3] Charles Price. MIPS IV Instruction Set. Revision 3.2. mips-isa.pdf. 1995.

[4] Dominic Sweetman. See MIPS run Linux. Vol. Second Edition. Morgan Kaufman. 2007.