MIPS Home Computer System

Student
Mostafa Abd El-Aziz

Supervisor
Prof. Dr. Layla Abou Hadid

Table of Contents

1 Introduction **5**

2 Architecture **7**

2.1 Central Processing Unit 7

2.1.1 Introduction 8

2.1.1.1 Load and Store Instructions 9

2.1.1.2 ALU Instructions 10

2.1.1.3 Jump and Branch Instructions 13

2.1.1.4 Miscellaneous Instructions 15

2.1.1.5 Control Instructions 16

2.1.1.6 Instruction Encoding Summary 17

2.1.1.7 Addressing Modes 19

2.1.1.8 Programmer-Visible Pipeline Effects 19

2.1.2 Registers 20

2.1.2.1 General-Purpose Registers 21

2.1.2.2 LO and HI Registers 21

2.1.2.3 Control Registers 22

2.1.3 Memory Management 27

2.1.3.1 MIPS Logical Address Space 29

2.1.3.2 Translation Look-aside Buffer (TLB) 30

2.1.3.3 Cache Memory 35

2.1.4 Exception Handling 35

2.1.4.1 Exception Cases 36

2.1.4.2 Exception Vectors 37

2.1.4.3 Exception Protocol 38

2.1.4.4 Precise Exceptions 39

List of Figures

2.1 MIPS-I instruction formats . . . 17
2.2 Instructions encoded by opcode field . . . 18
2.3 Instructions encoded by function field when opcode field = *SPECIAL* . . . 18
2.4 Instructions encoded by rt field when opcode = *REGIMM* . . . 19
2.5 All possible addressing modes . . . 19
2.6 Conventional names/uses of MIPS registers . . . 21
2.7 Control registers . . . 22
2.8 Format of Index register . . . 22
2.9 Format of EntryLo register . . . 23
2.10 Format of BadVaddr register . . . 23
2.11 Format of EntryHi register . . . 24
2.12 Format of Status register . . . 24
2.13 Format of Cause register . . . 26
2.14 ExcCode values with their meaning . . . 26
2.15 Format of EPC register . . . 26
2.16 Memory access levels . . . 27
2.17 Process logical address spaces and mappings to physical addresses . . . 28
2.18 Address translation using page table . . . 31
2.19 Address translation using two-level page table scheme . . . 32
2.20 Format of TLB entry . . . 33
2.21 Excetion vectors for various exception types . . . 38

Chapter 1

Introduction

Chapter 2

Architecture

2.1 Central Processing Unit

The CPU module implements MIPS-I instruction set architecture along with additional instructions that control the behavior of the processor. In this section we present detailed description for the programming interface of the CPU, followed by a description for the implementation of that interface.

Before going into details, we shall define two keywords that will be used throughout the section:

- **Architecture**: The behaviour of the processor as seen by the programmer. This includes the instruction set, their opcodes and formats, architectural registers, interrupt/exception protocols, and other behavioral concerns that will be seen later in the section. The architecture describes all that can be abstracted from the implementation of the processor. Other authors may call this an instruction set architecture (ISA) or macro-architecture.
- **Micro-architecture**: How the architecture of the processor is implemented in switching circuits. We use this keyword to refer to our VHDL implementation for the architecture described in this section. There may exist several different implementations for the same architecture. The only restriction on the micro-architecture of a processor is to behave exactly as described by the architecture, but the implementation itself doesn't matter. For example, the architecture might guarantee that the 'effect' of the execution of a set of instructions will be in order, however, the processor might execute the instructions in an out of order fashion (OoOE) yet effectively behaves the same as described by the architecture.

The following subsections are divided into two sets corresponding to the two titles described above; Until subsection 5 (inclusive), we are just talking about the architecture. It's not before subsection 6 where we start to show how we implement the behaviour described in earlier subsections. Programmers can skip subsections 6, 7, 8, and 9.

Here we summarize what the subsections are about (SS# = subsection #):

SS1. an introduction to the basics of the architecture of our CPU.

SS2. detailed description for the architectural registers.

SS3. interrupt and exception handling.

SS4. paging and translation look-aside buffer (TLB).

SS5. caching as seen by the programmer.

SS6. introduction to the VHDL implementation of our CPU.

SS7. VHDL implementation for the pipeline.

SS8. VHDL implementation for interrupts and exceptions.

SS9. VHDL implementation for the cache and TLB.

2.1.1 Introduction

CPU instruction (also called order, operation) is the basic unit of execution. The main task of a processor is to execute a sequence of instructions (i.e, program) which shall have effect on the state of the processor and devices connected with the processor.

Following Von Neumann's model, the program is stored in memory along with variables which are accessible and modifiable by the program. Memory is an array of words, where a word in MIPS is 32 bits. Every instruction in MIPS is encoded in a single word that contains the opcode and the operands of the instruction.

Every word in memory has an address, that also means that every instruction in memory has an address called 'instruction address'. CPU maintains a register that stores the instruction address of next instruction to fetch, called program counter (PC). When an instruction is fetched, program counter is increased by 4 (number of bytes in word) without touching lower 2 bits (they are always zero), a behaviour which implies that:

- Unless there is a branch, a jump, or an exception, instructions are always executed in sequence, as opposed to machines with one-plus-one addressing scheme like RPC-4000 where a program could be scattered over the drum.
- Instructions should always be stored in 4-byte aligned words.

It is important to note that MIPS-I instruction set is a proper subset of the instruction set of our processor; meaning that the processor supports additional instructions and registers that control the behaviour of the processor as will be seen below.

CPU instructions are divided into the following groups:

1. Load and Store
2. ALU
3. Jump and Branch
4. Miscellaneous
5. Control

2.1.1.1 Load and Store Instructions

This group holds all instructions related to memory access, either read/load or write/store accesses. Every instruction word contains the following fields:

- **Opcode**: specifies that this instruction is a memory instruction, along with whether this instruction is a byte, half-word, or a word instruction, and whether this instruction is an aligned/unaligned, load/store, and signed/unsigned instruction (6 bits).
- **Source/destination register**: If the instruction is a load instruction, this is the address of the register where the value loaded from memory will be stored. If the instruction is a store instruction, this is the address of the register where the value stored to memory will be loaded. Since there are 32 data registers specified by MIPS ISA, this field takes a value from 0 to 31 (5 bits).
- **Base register**: address of data register that contains the base memory address. Just like source/destination register field, this field takes a value from 0 to 31 (5 bits).
- **Index immediate**: A 16-bit signed integer that is extended to 32-bit signed integer then added to the value of the base register. Result is the memory address that will be accessed.

This instruction group has a total of 12 instructions, each instruction has a unique value for the opcode field:

1. **LB**: Load Byte
2. **LBU**: Load Byte Unsigned
3. **SB**: Store Byte
4. **LH**: Load Halfword
5. **LHU**: Load Halfword Unsigned
6. **SH**: Store Halfword
7. **LW**: Load Word
8. **SW**: Store Word
9. **LWL**: Load Word Left
10. **LWR**: Load Word Right
11. **SWL**: Store Word Left
12. **SWR**: Store Word Right

The detailed behaviour of those instructions is described in the appendix.

As described above, the target memory address is calculated by adding the extended 32-bit signed immediate to the value of the base register, forming a 32-bit memory address. This implies that memory address space of MIPS-I instruction set is 4GB. There are two characteristics that apply to the address field:

- **Alignment**: All instructions (except LWL, LWR, SWL, and SWR) are aligned. This means that LW and SW target address must be a multiple of 4, while LH, LHU, and SH target address must be a multiple of 2.
- **Endianness**: Our processor is little endian; the least significant byte of a word in memory is the byte having the lowest address, and the most significant byte is the byte having the highest address.

2.1.1.2 ALU Instructions

This group contains all instructions related to arithmetic and logical operations. We divide them into 4 categories, according to how they are encoded:

1. ALU instructions with an immediate operand
2. ALU instructions with three operands
3. Shift instructions
4. Multiply and divide instructions.

We call the last three groups together as 'R-TYPE ALU instructions' for reasons illustrated below. In the next lines we describe the instruction groups in detail.

- **ALU Instructions With Immediate Operand:**

  The reader may expect that the first group (ALU instructions with immediate operand) is encoded exactly the same way as memory instructions. Both groups are included in the same encoding class (called I-TYPE) as will be seen later in the encoding subsubsection. Just like memory instructions, an ALU instruction with an immediate operand shall consist of the following fields:

  - **Opcode**: specifies that this instruction is an ALU with immediate operand instruction, along with the type of operation. (6 bits)
  - **Source register**: except for LUI, all operations in this group are binary operations, thus this field contains the address of the register that contains the value of the first parameter of the binary operation. For LUI instruction, this field must be zero. (5 bits)
  - **Destination register**: address of data register where the result of the operation will be stored. (5 bits)

- **Index immediate**: except for ANDI, ORI, XORI, and LUI instructions, this field is a 16-bit signed integer that is extended to 32-bit signed integer then used as the second parameter for the binary operation. For ANDI, ORI, and XORI, this field is a 16-bit unsigned integer that is extended to 32-bit unsigned integer then used as the second parameter for the binary operation. For LUI, this field is just copied into the highest 16 bits of the destination register.

Following is a listing for all ALU with immediate operand instructions specified by MIPS-I ISA, with every instruction having a unique value for the opcode field:

1. **ADDI**: Add Immediate Word
2. **ADDIU**: Add Immediate Unsigned Word
3. **SLTI**: Set on Less Than Immediate
4. **SLTIU**: Set on Less Than Immediate Unsigned
5. **ANDI**: And Immediate
6. **ORI**: Or Immediate
7. **XORI**: Exclusive Or Immediate
8. **LUI**: Load Upper Immediate

The detailed behaviour of those instructions is described in the appendix.

- **R-TYPE ALU Instructions:**

The remaining three groups of ALU instructions (ALU instructions with three operands, shift instructions, and multiply/divide instructions) are all encoded the same way which is shown below. All instructions encoded that way are called R-TYPE instructions. We will summarize all encoding classes later in this section.

- **Opcode**: always zero. (6 bits)
- **Source register 1**: except for SLL, SRL, SRA, SLLV, SRLV, SRAV, MFHI, MTHI, MFLO, and MTLO instructions, this field contains the address of the register that holds the value of the first parameter for the binary operation. For SLLV, SRLV, and SRAV, this field contains the address of the register that holds the amount of shift. For SLL, SRL, SRA, MFHI, and MFLO instructions, this field must be zero. For MTHI and MTLO instructions, this field contains the address of the register that holds the value that will be moved into the HI/LO register. (5 bits)
- **Source register 2**: except for SLL, SRL, SRA, SLLV, SRLV, SRAV, MFHI, MTHI, MFLO, and MTLO instructions, this field contains the address of the register that holds the value of the second parameter for the binary operation. For SLL, SRL, SRA, SLLV, SRLV, and SRAV instructions, this field contains the address of the register that holds the value to be shifted. For MFHI, MTHI, MFLO, and MTLO instructions, this field must be zero. (5 bits).

- **Destination register**: except for MULT, MULTU, DIV, DIVU, MFHI, MTHI, MFLO, and MTLO instructions, this field contains the address of the data register where the result of the binary operation or shift operation will be stored. For MFHI and MFLO instructions, value stored in LO/HI register will be moved into the register addressed by this field. For MULT, MULTU, DIV, DIVU, MTHI, and MTLO instructions, this field must be zero, since the result of these instructions is stored directly in LO register, HI register, or both. (5 bits)
- **Shift amount**: for SLL, SRL, and SRA instructions, this field simply contains the amount of shift. For all other instructions, this field must be zero. (5 bits)
- **Function code**: An extension to the opcode field, contains a unique code that defines the instruction to be performed. (6 bits).

With a unique function code for every instruction (and an opcode of 0), ALU instructions with 3 operands group includes the following instructions:

1. **ADD**: Add Word
2. **ADDU**: Add Unsigned Word
3. **SUB**: Subtract Word
4. **SUBU**: Subtract Unsigned Word
5. **SLT**: Set on Less Than
6. **SLTU**: Set on Less Than Unsigned
7. **AND**: And operation
8. **OR**: Or operation
9. **XOR**: Exclusive Or operation
10. **NOR**: Nor operation

The third group (shift instructions) contains the following instructions:

1. **SLL**: Shift Word Left Logical
2. **SRL**: Shift Word Right Logical
3. **SRA**: Shift Word Right Arithmetic
4. **SLLV**: Shift Word Left Logical Variable
5. **SRLV**: Shift Word Right Logical Variable
6. **SRAV**: Shift Word Right Arithmetic Variable

Finally, following is a listing for all instructions in the last group (multiply/divide instructions):

1. **MULT**: Multiply Word
2. **MULTU**: Multiply Unsigned Word
3. **DIV**: Divide Word

4. **DIVU**: Divide Unsigned Word
5. **MFHI**: Move From HI
6. **MTHI**: Move To HI
7. **MFLO**: Move From LO
8. **MTLO**: Move To LO

For detailed description for the behaviour of all of the instructions listed above, please refer to the appendix.

2.1.1.3 Jump and Branch Instructions

MIPS-I architecture defines a number of operations that control the path of execution. Jump instruction shall unconditionally modify PC register, while branch instructions modify PC register only when the condition specified within the instruction word is tautology.

Jump and branch instructions are divided into four groups:

1. Jump Instructions Jumping Within a 256MB Region
2. Jump Instructions to Absolute Address
3. PC-Relative Conditional Branch Instructions Comparing 2 Registers
4. PC-Relative Conditional Branch Instructions Comparing Against Zero

Next we illustrate every group in detail.

- **Jump Instructions Jumping Within 256MB Region:**

  The way the instructions of the first group (jump instructions jumping within 256MB region) are encoded is different from I-TYPE and R-TYPE. It's called J-TYPE and consists of the following fields:

  - **Opcode**: specifies that the instruction is a jump instruction of J-TYPE, along with whether the instruction should alter the link register or not. (6 bits)
  - **Instruction Index**: a 26-bit integer that is multiplied by 4 to form the lowest 28-bit of the new value of PC register. The highest 4 bits are taken from the old value of PC, that's why the instructions of this group are said to jump within 256MB region (4GB/16).

  This group only contains two instructions:

  1. **J**: Jump
  2. **JAL**: Jump and Link

- **Jump Instructions to Absolute Address:**

  Second group (jump instructions to absolute address) consists of instructions similar to instructions of the first group, yet have the ability to jump to any memory location, not just locations inside the current 256MB region. These instructions are encoded as R-TYPE like most ALU instructions, thus the instruction word shall consist of the following fields:

  - **Opcode**: always zero. (6 bits)
  - **Source register 1**: contains address of the register that holds the target 32-bit address. (5 bits)
  - **Source register 2**: must be zero. (5 bits)
  - **Destination register**: for JALR instruction, this field contains the address of the register where return address shall be stored. For JAL instruction, this field should be zero.
  - **Shift amount**: must be zero. (5 bits)
  - **Function code**: An extension to the opcode field, contains a unique code that defines the instruction to be performed (only jump or jump with link). (6 bits).

  Analogous to the first group, this group consists of these two instructions:

  1. **JR**: Jump Register
  2. **JALR**: Jump and Link Register

- **Conditional Branch Instructions Comparing 2 Registers:**

  The remaining two groups are concerned with conditional branching. The third group consists of instructions that compare two registers, then jump if the comparison yields a specific result, while the last group compare one register against zero, and jump if the comparison yields a specific result.

  The third group is just encoded as I-TYPE (like ALU immediate instructions and load/store instructions). The fields of the instruction word are interpreted as following:

  - **Opcode**: specifies that this instruction is a conditional branch instruction comparing 2 registers, along with the type of comparison to be performed. (6 bits)
  - **Source register**: this field contains the address of the register that contains the value that will be used as the first operand for comparison. (5 bits)
  - **Destination register**: except for BLEZ and BGTZ instructions, this field contains the address of the register that contains the value that will be used as the second operand for comparison. For BLEZ and BGTZ instructions, this field must be zero. (5 bits)

- **Index immediate**: a 16-bit signed integer that is extended to a 32-bit signed integer, then added to PC to form the effective address. Thus this group of instructions are 'PC-relative'.

The group consists of the following instructions:

1. **BEQ**: Branch on Equal
2. **BNE**: Branch on Not Equal
3. **BLEZ**: Branch on Less Than or Equal to Zero
4. **BGTZ**: Branch on Greater Than Zero

- **Conditional Branch Instructions Comparing Against Zero:**

The fourth group is also encoded as I-TYPE instructions, however, the fields shall have special values and meanings:

- **Opcode**: always one. (6 bits)
- **Source register**: this field contains the address of the register that contains the value that will be compared against zero. (5 bits)
- **Destination register**: used as an extension to the opcode field. This field shall contain a unique identifier for the operation to be performed. (5 bits)
- **Index immediate**: a 16-bit signed integer that is extended to a 32-bit signed integer, then added to PC to form the effective address. Thus this group of instructions are 'PC-relative'.

Finally, this last group consists of the following instructions:

1. **BLTZ**: Branch on Less Than Zero
2. **BGEZ**: Branch on Greater Than or Equal Zero
3. **BLTZAL**: Branch on Less Than Zero and Link
4. **BGEZAL**: Branch on Greater Than or Equal Zero and Link

For detailed description for the behaviour of all of the instructions listed above, please refer to the appendix.

2.1.1.4 Miscellaneous Instructions

This group of instruction consists of instructions used to trigger exceptions by software. In MIPS-I ISA, this type of instructions is called 'exception instructions'.

When the CPU encounters one of the instructions in this group, it pauses execution of the current program and transfers control to the exception handler. This behaviour is similar to software interrupts in x86 architecture.

The format of this class of instructions is similar to R-TYPE, with the 'code' field replacing source register, target register, destination register, and shift amount fields:

- **Opcode**: always zero. (6 bits)
- **Code**: available for use as a software parameter to be examined by the exception handler by loading the instruction word into a register.
- **Function code**: An extension to the opcode field, contains a unique code that defines the instruction to be performed (only break or system call). (6 bits).

This group consists of the following instructions:

1. **SYSCALL**: System Call
2. **BREAK**: Breakpoint

For detailed description for the behaviour of all of the instructions listed above, please refer to the appendix.

2.1.1.5 Control Instructions

MIPS-I architecture provides an ability to define coprocessor instructions in the instruction address space. The architecture can contain up to 4 groups of coprocessor instructions. MIPS-I assigns system control purposes to coprocessor 0, thus we allocate coprocessor 0 instructions for system control. Coprocessor instructions are specified by system designer, not MIPS-I ISA itself.

In this document, we shall use the keyword 'control instructions' to refer to this group of instructions. Control instructions are R-TYPE instructions with the following fields:

- **Opcode**: always 16. (6 bits)
- **Source register 1**: used to specify type of instruction. (5 bits)
- **Source register 2**: for MFC0 and MTC0 instructions, this field specifies the address of the general-purpose register whose value will be transferred to a control register, or altered by the value of the control register specified by 'destination register' field. For other instructions, this field must be zero. (5 bits)
- **Destination register**: for MFC0 and MTC0 instructions, this field specifies the address of the control register whose value will be transferred to or altered by the general-purpose register specified in the previous field. For other instructions, this field must be zero. (5 bits)
- **Shift amount**: must be zero. (5 bits)
- **Function code**: used, along with source register 1 field, to specify the type of the instruction. (6 bits)

The following list shows the instructions contained in this group:

1. **MFC0**: Move From Control Register

2. **MTC0**: Move To Control Register

3. **TLBR**: Read TLB Entry at Index

4. **TLBWI**: Write TLB Entry at Index

5. **RFE**: Return From Exception

For detailed description for the behaviour of all of the instructions listed above, please refer to the appendix.

2.1.1.6 Instruction Encoding Summary

As seen above, all instructions are encoded using either R-TYPE, I-TYPE, or J-TYPE format. Figure 2.1 summarizes the three formats and their corresponding fields. Note that field use/meaning depends on the encoded instruction itself, not the format.

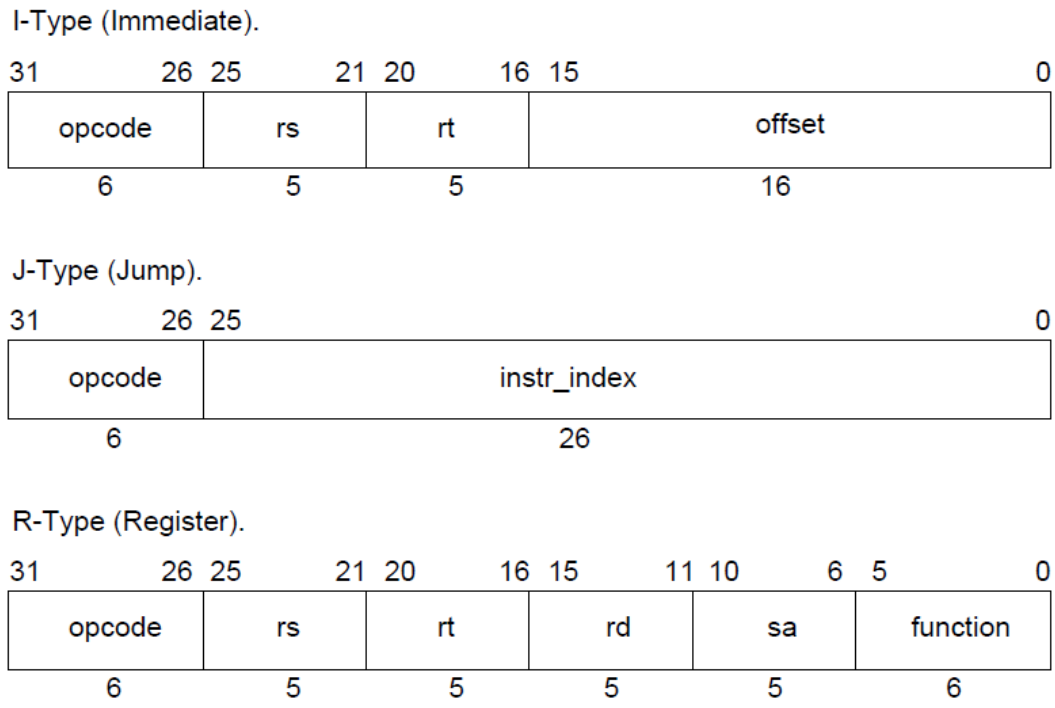
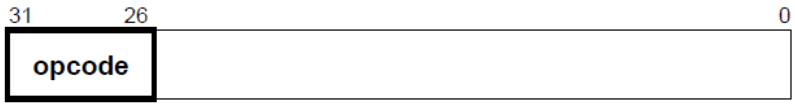


Figure 2.1: MIPS-I instruction formats

In all J-TYPE and I-TYPE instructions (except branch instructions comparing one register against zero), the opcode field is used to specify instruction type. This class of instructions is called 'instructions encoded by opcode field'. All possible values for opcode field under this class are shown in figure 2.2, along with valid mnemonics.

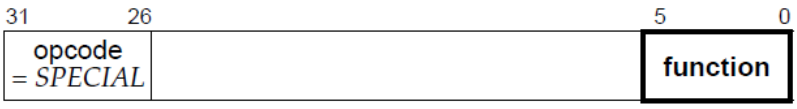


opcode bits 28..26 Instructions encoded by **opcode** field.

bits 31..29		0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	*SPECIAL* δ	*REGIMM* δ	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	010	*COP0* δ,π	*COP1* δ,π	*COP2* δ,π	*COP3* δ,π,κ	*	*	*	*
3	011	*	*	*	*	*	*	*	*
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	*
5	101	SB	SH	SWL	SW	*	*	SWR	*
6	110	*	LWC1 π	LWC2 π	LWC3 π,κ	*	*	*	*
7	111	*	SWC1 π	SWC2 π	SWC3 π,κ	*	*	*	*

Figure 2.2: Instructions encoded by opcode field

All R-TYPE instructions (including control instructions) are encoded using funct field, where opcode field is constant. For control instructions, the opcode field is equal to *COP*0 constant, while opcode field of remaining instruction is equal to *SPECIAL* constant. R-TYPE instructions with opcode equal to *SPECIAL* constant form another third class, called 'instructions encoded by function field when opcode field = *SPECIAL*'. Figure 2.3 shows members of this class.



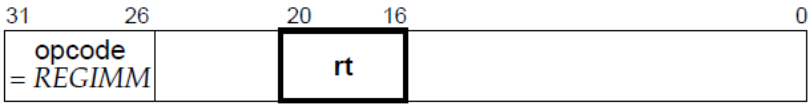
function bits 2..0 Instructions encoded by **function** field when opcode field = SPECIAL.

bits 5..3		0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1	001	JR	JALR	*	*	SYSCALL	BREAK	*	*
2	010	MFHI	MTHI	MFLO	MTLO	*	*	*	*
3	011	MULT	MULTU	DIV	DIVU	*	*	*	*
4	100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	101	*	*	SLT	SLTU	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	*

Figure 2.3: Instructions encoded by function field when opcode field = *SPECIAL*

Finally, branch instructions comparing one register against zero are encoded using rt field, while opcode field hold a constant value equal to *REGIMM* constant (register-immediate). This class, shown in figure 2.4, is called 'instructions encoded by the rt field

when opcode field = *REGIMM*'.



rt	bits 18..16	Instructions encoded by the rt field when opcode field = REGIMM.							
bits 20..19		0	1	2	3	4	5	6	7
		000	001	010	011	100	101	110	111
0	00	BLTZ	BGEZ	†	†	†	†	†	†
1	01	†	†	†	†	†	†	†	†
2	10	BLTZAL	BGEZAL	†	†	†	†	†	†
3	11	†	†	†	†	†	†	†	†

Figure 2.4: Instructions encoded by rt field when opcode = *REGIMM*

2.1.1.7 Addressing Modes

Figure 2.5 summarizes all possible addressing modes along with instructions that support them:

Addressing Mode	Corresponding Instructions
Register, [Base Register]+Immediate Offset	Load/Store Instructions
Register, Register, Immediate	ALU Instructions with Immediate Operand
Register, Register, Register	ALU Instructions with Three Operands
	SLLV, SRLV, and SRA
Register, Register, Shift Amount	SLL, SRL, and SRA
Register, Register	MULT, MULTU, DIV, and DIVU
Register	MFHI, MTHI, MFLO, and MTLO
Absolute Address Within 256MB Segment	Jump Instructions Jumping Within 256MB Region
Register Absolute	Jump Instructions to Absolute Address
Register, Register, PC-Relative Immediate	Branch Instructions Comparing 2 Registers
Register, PC-Relative Immediate	Branch Instructions Comparing Against Zero
Immediate Code	Exception Instructions
Register, Control Register	MFC0 and MTC0
Implicit	TLBR, TLBWI, and RFE

Figure 2.5: All possible addressing modes

We note that every instruction has no more than one single addressing mode, thus MIPS doesn't need an addressing mode field in instruction word.

2.1.1.8 Programmer-Visible Pipeline Effects

Several effects related to the behaviour of the pipeline should be stated clearly so the programmer can be aware of the exact performance parameters of the running program:

- **Delayed branching**: Since the effective address is not available immediately after fetching a jump/branch instruction, a delay slot should be inserted after the

jump/branch instruction. The delay instruction is always fetched and executed after the jump/branch instruction. The effective address becomes ready after fetching the delay instruction and next instruction would be the instruction at the new PC.

- **Load data not available**: If a memory load instruction is followed by an ALU instruction that requires the data acquired by the load instruction (addresses the same register), a data hazard will occur since ALU must wait until MEM unit is done fetching data from memory. This shall delay the execution of the ALU instruction by one delay slot (stalls the pipeline for one slot).

- **Consequent control instructions**: A control instruction may stall if there is another control instruction in the pipeline, to prevent data hazards and simultaneous access to TLB and stuff.

- **Branch instruction operands not ready**: If a branch instruction depends on result of the previous instruction (or second previous instruction), the pipeline will stall until data is ready.

- **Multiply/divide instructions**: These four special instructions take more than one CPU clock cycle to complete, therefore they might stall the pipeline.

- **Cache misses**: The pipeline shall stall if the instruction to be fetched or data specified by load instruction is not in cache. Furthermore, store instructions would always stall the pipeline since the cache system is write-through. This means that an instruction will take more than one CPU clock cycle if a cache miss happens.

- **Unaligned memory access**: This class of instructions shall take more than one CPU clock cycles because atomic access to unaligned word is not possible.

Other than the cases specified above, the pipeline will have no hazards/stalls and every instruction will take no more than one clock cycle delay.

2.1.2 Registers

A register is a small amount of memory that can be accessible faster than memory/cache. Processor provides direct register access through 'register' addressing mode.

MIPS architecture defines number of registers to be used as general-purpose registers by the running software. MIPS architecture also defines two special-purpose registers called LO and HI registers.

In addition to registers defined by MIPS standard, our processor contains special registers that control the behaviour of the processor, called 'control registers'. Surprisingly, control registers are accessible through control instructions.

2.1.2.1 General-Purpose Registers

Strictly speaking, MIPS architecture doesn't make any assumptions about the use of the 32 general-purpose registers. It's up to the programmer to use the register the way she likes, and the hardware will just perform the operations on those registers as specified by the programmer. However, there are two exceptions to this:

1. Register 0 is always zero, no matter what value the programmer writes to it.
2. Register 31 is used as a link register (register containing return address from subroutine). Some instructions implicitly address this register.

Although MIPS architecture doesn't make any assumptions on the use of general-purpose registers (except for register 0 and 31), people used to use those registers according to conventions listed in figure 2.6. The figure shows conventional names and uses associated with the registers. Available assemblers and compilers for MIPS architecture are expected to follow the same conventions.

Reg#	Name	Description
0	zero	Always zero
1	at	Used by assembler (*at* = *assembler temporary*)
2-3	v0-v1	Value returned by subroutine
4-7	a0-a3	Subroutine parameters (*a#* = *argument#*)
8-15	t0-t7	Used by subroutines without saving (*t#* = *temporary#*)
24-25	t8-t9	
16-23	s0-s7	Must be saved before use (*s#* = *subroutine_var#*)
26-27	k0-k1	Used by exception handler
28	gp	Global pointer
29	sp	Stack pointer
30	s8/fp	9th register variable/frame pointer
31	ra	Return address

Figure 2.6: Conventional names/uses of MIPS registers

2.1.2.2 LO and HI Registers

Along with the 32 registers defined above, two special registers are defined by the MIPS architecture to store results of MULT, MULTU, DIV, and DIVU instructions. Since these four instructions produce 64-bit results, they need two destination registers. Since MIPS architecture doesn't define more than one destination register for an instruction (for hardware considerations), the result should first stored in an intermediate 64-bit register (or two parallel 32-bit registers) then moved to the GPRs in two steps.

The intermediate registers are called LO and HI registers. Each one of them is 32-bit wide, so you can see them as one 64-bit register which is can be accessed in halfs. LO and HI can be read using MFLO and MFHI instructions, each of which copies the value of LO or HI register into a general-purpose register. The HI and LO registers can also altered by MTLO and MTHI instructions.

2.1.2.3 Control Registers

Now we turn our attention to the last class of registers, registers that control the behaviour of the processor. These registers are not defined by MIPS architecture itself, yet a processor needs to have a set of registers to store control switches. These registers are analogous to CR0, CR2, CR3, and EFLAGS registers of x86 architecture.

Control registers are accessible only through MFC0 and MTC0 instructions, which transfer values between GPRs and control registers. Every control register is addressed by a special index. Figure 2.7 shows all the control registers of our processor and their uses.

Reg Indx	Name	Description
0	Index	Stores index of TLB entry to be read/written
2	EntryLo	The lowest half of TLB entry to be read/written
9	BadVaddr	Stores the memory address that caused TLB miss
10	EntryHi	The highest half of TLB entry to be read/written
12	SR	Status register
13	Cause	Exception type
14	EPC	Address of the instruction that caused an exception

Figure 2.7: Control registers

Every control register has a specific format with a special function for every field. Next we show the format of every control register with a detailed description for the fields and how they can be used.

- **Index Register:**

Index Register (0)				
Field bit(s):	31	30..14	13..8	7..0
Field name:	0	x	Index	x

Figure 2.8: Format of Index register

Detailed description for fields in figure 2.8:

  - **Index**: Before the operating system reads or modifies a TLB entry, it shall first write the index for the target TLB entry in this field using MTC0 instruction. After then, the OS issues a TLB read or TLB write using TLBR or TLBWI instruction. Index takes a value from 0 to 63, which implies that our TLB has 64 entries.

- **EntryLo Register:**

EntryLo Register (2)				
Field bit(s):	31..12	11..10	9	8..0
Field name:	PFN	0	V	0

Figure 2.9: Format of EntryLo register

Detailed description for fields in figure 2.9:

  - **V**: This field matches the valid bit (V) of the TLB entry to be read or written by the OS. Before the OS writes a TLB entry, it writes the valid bit of the entry in this field. When TLBWI instruction is issued, the V field of EntryLo register is copied into the V field of TLB entry addressed by Index register. On the other hand, when a TLBR instruction is executed, the V field of TLB entry addressed by Index register is copied into V field of EntryLo register; a read for EntryLo.V field (using MFC0 instruction) reveals what the value of the V field of the addressed TLB entry was when TLBR was issued.
  - **PFN**: This field matches the physical frame number (PFN) field of the TLB entry to be read or written by the OS. Before the OS writes a TLB entry, it writes the PFN of the entry in this field. When TLBWI instruction is issued, the PFN field of EntryLo register is copied into the PFN field of TLB entry addressed by Index register. On the other hand, when a TLBR instruction is executed, the PFN field of TLB entry addressed by Index register is copied into PFN field of EntryLo register; a read for EntryLo.PFN field (using MFC0 instruction) reveals what the value of the PFN field of the addressed TLB entry was when TLBR was issued.

- **BadVaddr Register:**

BadVaddr Register (9)	
Field bit(s):	31..0
Field name:	BadVaddr

Figure 2.10: Format of BadVaddr register

Detailed description for fields in figure 2.10:

  - **BadVaddr**: When a TLB miss happens, the memory address that causes the TLB miss is copied in this field. There are two cases here:
    1. If a TLB miss happens while fetching an instruction from memory, the address of the instruction is copied into BadVaddr field before the exception handler is called.

2. If the instruction is fetched successfully, but it happens that the instruction is a load/store instruction, and a TLB miss happens during fetching/writing data from/to memory, the effective address of the data element addressed by this instruction is placed in BadVaddr before the exception handler is called.

- **EntryHi Register:**

EntryHi Register (10)		
Field bit(s):	31..12	11..0
Field name:	VPN	0

Figure 2.11: Format of EntryHi register

Detailed description for fields in figure 2.11:

- **VPN**: This field matches the virtual page number (VPN) of the TLB entry to be read or written by the OS. Before the OS writes a TLB entry, it writes the VPN field of the entry in this field. When TLBWI instruction is issued, the VPN field of EntryHi register is copied into the VPN field of TLB entry addressed by Index register. On the other hand, when a TLBR instruction is executed, the VPN field of TLB entry addressed by Index register is copied into VPN field of EntryHi register; a read for EntryHi.VPN field (using MFC0 instruction) reveals what the value of the VPN field of the addressed TLB entry was when TLBR was issued.

- **Status Register:**

Status Register (12)						
Field bit(s):	31..5	4	3	2	1	0
Field name:	0	IEo	0	IEp	0	IEc

Figure 2.12: Format of Status register

Detailed description for fields in figure 2.12:

- **IEc**: Interrupt enable/disable flag. When software writes 1 to this field, interrupts are enabled. When software writes to this field, interrupts are disabled and do not trigger any exception. This field is reset to 0 by hardware on two events:
  1. CPU is reset.
  2. An exception has taken place.

This field is also modified by RFE instruction which copies the value of IEp field into IEc field.

It's important to note that when an exception happens and the value of IEc is reset to 0, the old value of IEc is not lost but rather is copied into IEp. Also, the old value of IEp is copied into IEo before IEc is copied into IEp. When exception handler returns, RFE instruction is executed and the value of IEp is copied back to IEc, and the value of IEo is copied back to IEp.

Actually, the three fields (IEc, IEp, and IEo) behave like a 3-level stack for the value of the interrupt flag. The top of the stack (IEc) is the current state of the interrupt flag. When an exception happens, a 0 is pushed into stack, causing interrupts to be disabled and the values of the three entries are shifted by 1 (IEp copied into IEo and IEc copied into IEp).

If a nested exception occurs again, the process repeats, thus IEo shall then hold the old value of the interrupt flag, IEp shall hold the value of the interrupt flag during handling the first exception, and IEc will be reset to 0.

When the exception handler returns, RFE instruction will cause the stack to pop; IEp will be copied back to IEc and IEo will be copied back to IEp, thus IEc returns back to its original state before the exception has occured.

This behaviour allows an exception to be nested into another exception without need to push status register into the software stack before the inner exception happens.

- **IEp**: As discussed above, this field functions as backup for IEc field during exception handling. The three fields (IEc, IEp, IEo) can be seen as a 3-level stack to store the value of the interrupt flag across exceptions.

  When an exception happens, the value of this field is copied into IEo field and the value of IEc is copied into this field before IEc is reset to 0. When an RFE instruction is executed, the value of this field is copied back to IEc and the value of IEo is copied into this field.

- **IEo**: As discussed above, this field functions as backup for IEp field during exception handling. The three fields (IEc, IEp, IEo) can be seen as a 3-level stack to store the value of the interrupt flag across exceptions.

  When an exception happens, the value of IEp is copied into this field before IEc is copied into IEp. When an RFE instruction is executed, the value of this field is copied back to IEp.

- **Cause Register:**

Cause Register (13)				
Field bit(s):	31	30..7	6..2	1..0
Field name:	BD	0	ExcCode	0

Figure 2.13: Format of Cause register

Detailed description for fields in figure 2.13:

- **ExcCode**: When an exception occurs, the type of the exception is copied into this field before the exception handler is called. Currently, only the types of exceptions shown in figure 2.14 are supported.

ExcCode Value	Description
0	Interrupt Exception
1	TLB Miss Exception

Figure 2.14: ExcCode values with their meaning

- **BD**: When an instruction causes exception (or gets interrutpted), this field is set to 1 if the instruction is a delay slot instruction, otherwise it is reset to 0; thus 'BD' means branch detected.

- **EPC Register:**

EPC Register (14)	
Field bit(s):	31..0
Field name:	EPC

Figure 2.15: Format of EPC register

Detailed description for fields in figure 2.15:

- **EPC**: When an instruction that is currently executed by the processor causes an exception (like TLB miss) or gets interrupted (by en external I/O device), the address of the instruction is copied into EPC field, then exception handler is called. If the instruction is a delay slot instruction, the address of the previous jump/branch instruction is copied into EPC field instead of the address of the delay slot instruction.

  This behaviour prevents software to misbehave when the exception handler returns (as the branch instruction needs to be executed again or the processor will keep fetching instructions that follow the delay slot no matter the branch is taken or not). This special case causes BD field of Cause register to set as

described above.

It's also important to note that before exception handler is called, the CPU makes sure that the instruciton doesn't have any effect on the architectural state of the processor (i.e, any effect caused by the instruction is undone).

2.1.3 Memory Management

Having discussed the instruction set and registers of our CPU, we now turn our attention to how the CPU translates memory accesses. When an instruction is fetched from memory, or when a load/store instruction requests data from memory, the memory address is first translated by means of translation look-aside buffer (TLB). The translated address is then used to look-up data in the cache which might then issue an access request to main memory if data is not found in cache. Figure 2.16 summarizes the process.

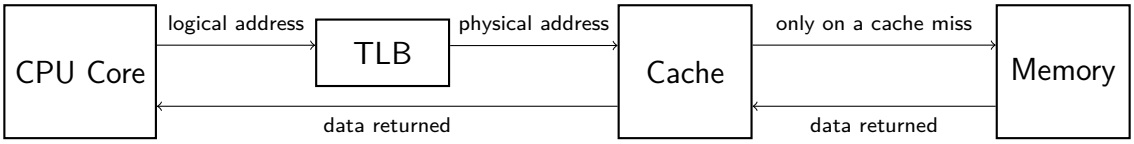


Figure 2.16: Memory access levels

The figure presents two different types of addresses: logical address and physical address. We define both terms as follows:

1. **Logical address**: the address of a memory location or an instruction from the prespective of the running program. This simply means that a program doesn't see memory as it is in reality. The running program doesn't need to know the real address of the memory item it is currently addressing, but the operating system must provide the mapping between the logical address of a memory item and its real location in memory. This mapping is used by translation circuitry that translates logical address to the real location before main memory is queried.

2. **Physical address**: this is the real location of the memory item or the instruction that is being accessed. Physical address is the address that is put on the bus when main memory access is established by the processor. Also, as will be seen later, data in cache is looked up by its physical address, not logical address.

The reason why there are two separate types of addresses and a translation circuitry is to allow every process to have its own address space by providing a distinct logical-to-physical address mapping for every running process. The process of translating a logical address to its corresponding physical address is called **address translation**.

Figure 2.17 illustrates how we can achieve process-level isolation using this technique. Every process has its own set of memory items: blue items belong to process 1, and red items belong to process 2. The OS needn't allocate the items in physical address space

in a specific order (they are scattered over memory) since the running program doesn't need to know the physical locations and doesn't need to make any assumptions about them.

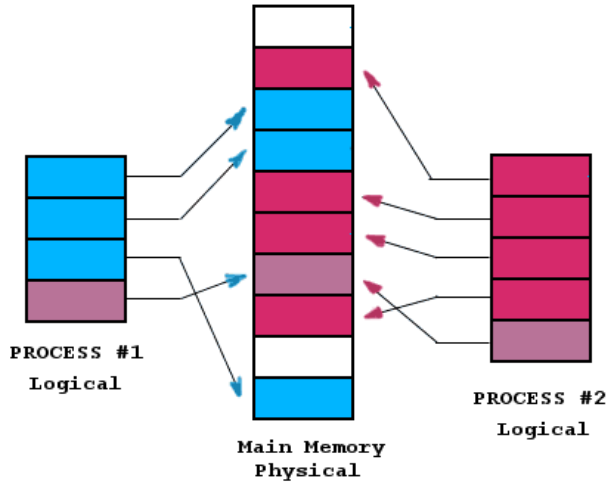


Figure 2.17: Process logical address spaces and mappings to physical addresses

The OS provides two separate mappings for both processes in the figure. Each mapping provides enough information needed to translate logical addresses of memory items belonging to the process to physical addresses. Each process only sees its own items in the logical address space, therefore a process can never access memory items of other processes because there is no mapping for them. By this mechanism, we have achieved the required isolation between processes: every process can only access the items allocated to it.

The OS might allow some memory items to be shared between more than one process. The purple item in the figure is seen in the logical address space of both process 1 and process 2. This implies that an item in main memory can have more than one logical address (for several mappings). However, every memory item can have no more than one physical address.

The process of allocating and organizing items in main memory is the responsibility of the OS. Also, the OS is responsible for providing the required mapping between logical and physical addresses. The OS provides the CPU with the mapping of the process that is currently being executed, and the CPU is responsible only for doing the translation on every memory access. Later in this section we show how the OS informs the CPU of the mapping.

It's important to note that some authors use terms like 'virtual address' and 'program address' to refer to logical addresses. We shall use the term 'logical address' throughout the document to prevent confusion with other terms and techniques, like 'virtual memory', which shouldn't be confused with 'logical address space', or equivalently, 'logical

memory'.

2.1.3.1 MIPS Logical Address Space

Since a logical address is 32 bits (as seen earlier), every process shall has a logical address space of 4GB. This 4GB address space is divided into four regions, each region is translated differently as will be seen below. The four regions are:

1. **kuseg** 0x00000000 - 0x7FFFFFFF (2GB): any address in this region is translated by the translation circuitry to its corresponding physical address using the mapping provided by the OS. This area usually contains the text and data of the running process.

2. **kseg0** 0x80000000 - 0x9FFFFFFF (512MB): addresses in this region are not translated using the mapping provided by the OS, but rather they are translated by stripping off the most significant bit. Thus, this region simply maps to the physical region: 0x00000000 - 0x1FFFFFFF. Operating system kernel can be simply put inside the lower 512MB zone of the main memory, and consequently it will appear inside kseg0. This implies that the kernel doesn't need to setup any mapping for itself, it is just mapped automatically to this region. This also implies that kernel memory area is shared among all processes.

3. **kseg1** 0xA0000000 - 0xBFFFFFFF (512MB): just like kseg0, addresses in this region also translate to the lower 512MB of physical address space. The only difference between kseg0 and kseg1 is that memory access of any address in kseg1 will skip the cache, so it is always uncached, while kseg0 is always cached. Access to memory-mapped I/O registers should always be done through this segment (to prevent caching). System designers must make sure that I/O registers have physical addresses in the low 512MB physical zone so that they can appear in kseg1.

4. **kseg2** 0xC0000000 - 0xFFFFFFFF (1GB): any address in this region is simply translated using the mapping provided by the operating system, this area is usually used to contain OS data structures which are maintained by the kernel.

We conclude that addresses in kuseg and kseg2 are translated the same way, using the mapping provided by the operating system, while addresses in kseg0 and kseg1 are translated by mapping them directly to the lower 512MB zone of physical memory. The only difference between kseg0 and kseg1 is that the former is cached while the later is uncached. The difference between kuseg and kseg2 is that kuseg is used to hold current process code and data, while kseg2 is used to hold kernel objects.

Having a static mapping for kseg0 and kseg1 has several advantages:

1. Processor reset vector is put in kseg1. Consequently, system initialization code (i.e, firmware) can start execution before translation circuitry is initialized. Caching also might not be wanted during system initialization, thus kseg1 is a safe place to start the system from.

2. Kernel is shared among all processes and doesn't need a specific mapping. This means that kernel initialization code can be executed before the translation circuitry is initialized. This also implies that a system call from process code running in kuseg can be handled in kseg0 without need to switch logical address space (i.e, without providing translation circuitry with another mapping). Microkernels can also use this area to put the tiny code that does the logical address switching.

When a process switch occurs, the OS must alter the translation circuitry with the mapping of the new process. The kernel can fix the mapping of kseg2 among all processes. Consequently, kernel objects can also be shared by all processes, allowing system calls to be handled without need to switching logical address space or alter the mapping.

2.1.3.2 Translation Look-aside Buffer (TLB)

Now we show how the operating system provides the CPU with a certain mapping for current process. Since there are 3GB of the address space (kuseg and kseg2 regions, see above) that can be dynamically mapped to physical memory, we need a data structure that describes how to map every logical address inside the two regions (kuseg and kseg2) into physical address.

One can think of an array of 4G elements, where the index of the array is the logical address, and the contents of an array element is the physical address that maps to the logical address associated with that entry. Notice that the third quarter of such an array would always be sparse, since kseg0 and kseg1 are not dynamically mapped as shown before.

Since a physical address is 32 bits, we need an array of 4Gx4B which is a 16GB data structure for every process. This is certainly unfeasible and we would need to think for another solution. Instead of mapping each logical address to a certain physical address, we can map every block of logical address to a block of physical addresses.

Using this technique, we just need an array with n elements, where n is number of blocks of the logical address space which is also equal to number of block of the address space. We call this scheme **'Paging'**, where a physical address is called a 'frame' and logical address block is called a 'page'.

Given a page size of 4KB, the 4GB address space could be divided into 1M pages, requiring an array of only 1M elements. The array is called a 'page table' and maps every page of the logical address space to a frame in the physical memory. A page table entry will simply contains the number of the frame it maps to, which is a 20-bit word (because we've got a total number of 1M frames).

Extra information might need to be stored along with frame number, like whether this page is mapped or not (a page can be unmapped if doesn't map to any physical address) as well as OS-specific flags. Hence we can extend the 20-bit entry to 32 bits: 12 bits for flags and 20 bits for frame number (FN). Hence we need an array of 1M*4B which is equal to 4MB page table for every process. This is much smaller than 16GB.

To translate a logical address, we divide the address into two parts:

1. Page number: the highest 20 bits of the address

2. Offset inside the page: the lowest 12 bits of the address.

Page number is then used to index the page table to get the corresponding page table entry. The frame number is extracted from the entry and concatenated with the 12-bit offset to form the actual physical address that this logical address maps to. Figure 2.18 illustrates this translation process.

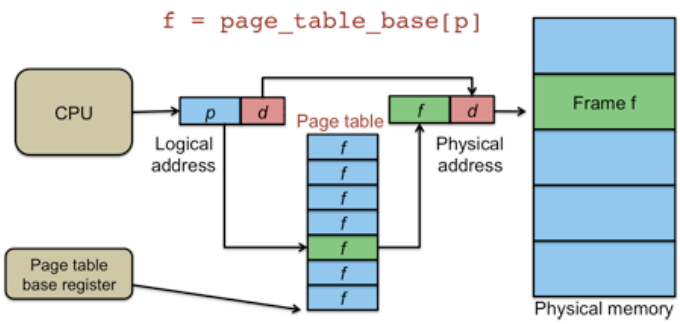


Figure 2.18: Address translation using page table

A page table of 4MB fo every process is still big, since when number of processes increase, a big portion of main memory will just be used to store page tables. Furthermore, a process won't usually need to map the whole 4GB address space. It can leave some pages unmapped, and only map pages where it needs to put its code and data. Actually, the usual case is that most pages are unmapped. And if virtual memory is not implemented, the total number of mapped pages across all page tables (across all processes) must be less than or equal to size of RAM (given that no frame is mapped to by more than one page), and RAM is usually much smaller than 4GB.

Since most entries in the page table are sparse (because most pages in the logical address space are unmapped), the OS can make use of this characteristic to make the size of the page table smaller. Thus two-level page tables where introduced. The big 1M-entry page table is divided into 1024 parts, with each part having 1024 entries. Each part is called a second-level page table. These second-level page tables could be scattered around memory. An 1024-entry table could be used to store the addresses of these 1024 2nd-level tables. Figure 2.19 shows how translation works under this two-level scheme.

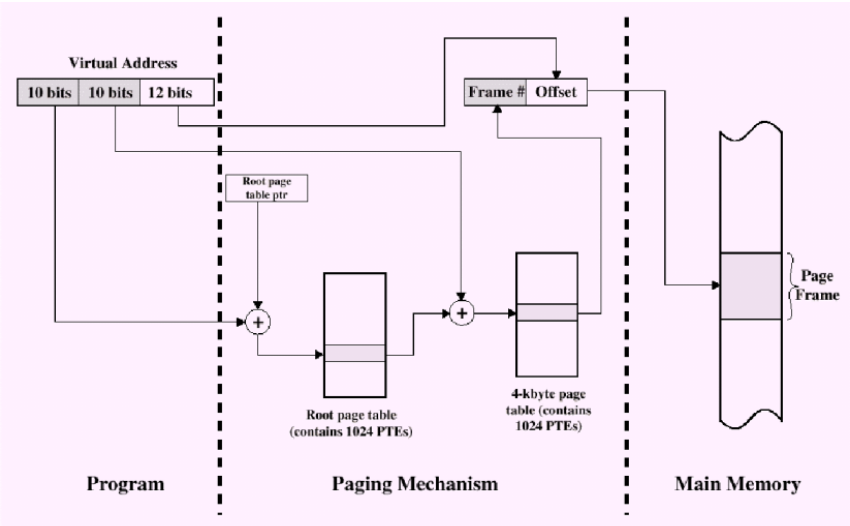


Figure 2.19: Address translation using two-level page table scheme

This way, if all the entries of a second-level page tables are unmapped, the whole second-level page table could be unmapped, setting a flag in the page directory that this page table doesn't exist at all, and all pages in this region are unmapped. Consequently, many second-level page tables wouldn't need to exist at all, reducing the size of allocated memory for page tables to a reasonable limit. However, if the process maps all of its logical memory, page table increases to 4MB plus the 4KB of the page directory.

This result implies that we cannot store the whole page table in the processor, since we would need 4MB of memory inside the CPU dedicated for page table. We would also need to re-initialize the whole 4MB on every process switch, a process that will take much time. That's why page tables are stored in memory and with a multi-level scheme to redcue their sizes.

So, how the CPU makes use of these page tables to do translation? The processor could simply be given the base physical address of the page directory that controls the current process. When a logical address is generated by core processor, the translation circuitry looks up the page directory then the corresponding 2nd-level page table to get frame number, as shown in the figure above. On a process switch, the CPU is given the page directory of next process.

This translation scheme requires two memory access for each translation, which is too much since memory access takes much time, so we need to cache page entries that are used frequently. Caching page entries will effectively reduce average translation time. This special-purpose cache is called **'translation look-aside buffer'**, or simply TLB. When a logical address is generated by core processor, TLB is queried for that logical address. If the entry of this address is found inside TLB, the physical address is returned immediately. If a TLB miss happens, the processor shall access main memory to get page entry.

The type of TLB described above is called **'hardware-managed TLB'**, where TLB is totally isolated from the OS and is managed by hardware. Another mechanism would be **'software-managed TLB'**, where TLB is controlled by the OS, and CPU doesn't need to know any information about page table. In a software-managed TLB, when a TLB miss happens, the CPU raises an exception, and it is the responsibility of the exception handler to look-up page tables and refill TLB with the missed entry.

Since our processor is designed for systems where resources are very limited (like FPGA), we chose to use software-managed TLB to simplify the implementation of the processor. On a process switch, the OS just needs to clear TLB from entries related to old process. After TLB is cleared, and when a logical address is generated, a TLB miss happens and exception handler is called. The OS queries page tables and refills TLB. Page tables are totally managed by the OS; the processor doesn't need to know anything about their structure and locations.

The TLB consists of 64 entries. Every entry is 64-bits wide with the structure illustrated by figure 2.20.

TLB Entry						
Field bit(s):	63..44	43..32	31..12	11..10	9	8..0
Field name:	VPN	0	PFN	0	V	0

Figure 2.20: Format of TLB entry

Fields are:

- **VPN**: the 20-bit logical page number to be translated.
- **PFN**: the 20-bit physical frame number this VPN maps to.
- **V**: valid bit. If this bit is set to 1, this is entry is valid and can be used for translation. If this bit is set to 0, this entry is not valid and won't be used for translation.

The reader can notice the similarity between TLB entry and EntryLo/EntryHi registers. The structure of the lower 32 bits of a TLB entry is identical to EntryLo register, and the structure of the upper 32 bits is identical to EntryHi register. EntryLo and EntryHi registers, along with Index register, are used to read and write TLB entries. For reading a TLB entry, the steps are:

1. The OS writes the index (0->63) of the TLB entry to be read into Index register using MTC0 instruction.
2. TLBR instruction is executed. The instruction causes the lower 32 bits of the TLB entry indexed by Index register to be copied into EntryLo register, and the upper 32 bits of the TLB entry to be copied into EntryHi register.
3. The operating system reads EntryLo register using MFC0 instruction. Result is the lower 32 bits of the target TLB entry.

4. The operating system reads EntryHi register using MFC0 instruction. Result is the upper 32 bits of the target TLB entry.

A similar sequence could be used for writing a TLB entry

1. The OS writes the index (0->63) of the TLB entry to be written into Index register using MTC0 instruction.
2. The operating system writes the lower 32 bits of the target TLB entry into EntryLo register using MTC0 instruction.
3. The operating system writes the upper 32 bits of the target TLB entry into EntryHi register using MTC0 instruction.
4. TLBWI instruction is executed. The instruction causes EntryLo register to be copied into the lower 32 bits of the TLB entry indexed by Index register, and EntryHi register to be copied into the upper 32 bits of the target TLB entry.

Now since the operating system has full access to TLB, paging can be implemented easily. Whenever a logical address is generated by the processor, the following steps happen:

1. The CPU reads TLB entry which has an index that is equal to bits 12 to 17 (inclusive) of the logical address.
2. If V flag of the read entry is 0, the CPU copies the logical address into BadVaddr register, and TLB miss exception happens. In this case translation stops. If V flag is 1, continue to step 3.
3. The VPN of the read field is compared with the highest 20 bits of the logical address. If both values are not equal, the logical address is copied into BadVaddr register and TLB miss exception happens. In this case translation stops. If both values are equal, continue to step 4.
4. The PFN of the read field is concatenated with the lower 12 bits of the generated logical address, the result is the physical address that will be used to access memory. Translation is done and the CPU continues work normally without exception.

Step 1 implies that our TLB is actually a **direct-mapped cache** where the index is computed using bits 12 to 17 of the logical address, and tag is VPN. Using this information, the OS can handle TLB miss exception as follows:

1. The OS reads the value of BadVaddr register using MFC0 instruction.
2. The OS then queries page tables to know the mapping of the address in BadVaddr register. If there is no mapping, the operating system may decide to stop the program or send an error signal for instance. If there is mapping, the operating system reads the physical frame number that this logical address maps to.
3. The operating system calculates the index of the TLB entry which should be modified to solve the miss. The index is actually equal to bits 12 to 17 (inclusive) of the logical address as explained above.

4. Now the OS has the index, VPN, and the corresponding PFN. The OS simply writes the index to Index register, writes VPN to EntryHi register, and writes PFN to EntryLo register (and sets V flag of EntryLo register to 1).

5. The OS executes TLBWI instruction. EntryLo and EntryHi registers are then copied into the TLB entry.

6. Exception handler returns and the instruction that causes the miss is executed again, but this time it doesn't cause a TLB miss and execution continues normally.

2.1.3.3 Cache Memory

After logical address is translated into physical address, physical address is used to access RAM. However, because memory access takes long time (i.e, has high penalty), a cache is put between TLB and RAM to fasten memory access time.

Since the pipeline of our processor might be fetching an instruction and handling memory access for a load/store instruction at the same time, two separate caches are employed: an instruction cache and a data cache. Both caches are direct-mapped caches. Every entry in the cache consists of three fields:

- **Valid flag**: If 1, data and tag fields are valid. Otherwise the entry is invalid.
- **Data field**: The 4-byte datum cached by this entry.
- **Tag field**: A 20-bit field storing the higher 20 bits of the physical address associated with this entry.

Every cache has 1024 entries, resulting in two 4KB caches. The process of looking up data in the cache is completely transparent from software. Bits 2 to 11 (inclusive) of the physical address are used as index. The tag of the entry at the calculated index is compared to the higher 20 bits of the physical address. If valid bit is 1 and comparison matches, data field of the entry is return to core processor. If valid bit or 0 or comparison doesn't match, the CPU initiates a memory cycle to bring data from RAM and updates the cache automatically.

Cache is write-through; any store instruction will cause data to be written to both cache and RAM. A cache miss (or a store instruction) causes the instruction to delay until data is brought from or written to memory. However, cache hit can be handled without any delay; pipeline doesn't stall and advance normally. Although caching is totally isolated from the programmer, the programmer can use her knowledge of how cache works to be able to write optimized programs that make full use of caching.

2.1.4 Exception Handling

In MIPS, an exception is a hardware event, trap, system call, or anything that interrupts the normal flow of the program and needs special handling. The same mechanism is used to handle all exceptions, which include:

- **Interrupts**: throughout this document, we will use the term 'interrupts' to refer only to those generated by hardware devices. Interrupts are external events that need immediate interaction by the processor. Interrupts can be enabled/disabled by software (refer to Status Register). If enabled, when an interrupt signal arrives to the processor, exception happens and the normal flow of the program is interrupted. Interrupts may include, for example, keystrokes, mouse button events, and disk I/O events.

- **System calls**: system calls are special calls that transfers the control to the operating system to execute some kernel-level subroutine then return back to user program and continue execution normally. System calls are handled by the processor as an exception trigger: when system call (or breakpoint) instruction is executed, an exception happens.

- **Page faults**: page faults need to be reported to operating system whenever they happen to fully support paging. When a page fault happens, the running program is interrupted and exception handler is called. Exception handler shall handle the fault then return back to the running program.

- **Program errors**: division erros, invalid opcodes, and invalid addresses are also forms of exceptions that cause the exception handler to be called. In this case the exception handler might, for example, call a specific signal handler in the faulty process or stop it completely.

Processor provide a mechanism to return from exception and resume execution of the interrupted process. It's the responsibility of the OS to store the context of the interrupted process at beginning of exception handling, so that when the exception handler returns back to the interrupted process, the process continues its work normally as if nothing has happened.

It's the responsibility of the processor to inform software with what kind of exception has actually happened, at which instruction the exception has happened, and any extra parameters needed. For example, if a program error has happened, the processor shall inform software of which instruction has caused the error. If a page fault happens, the processor should inform software of which instruction caused the fault, and of the logical address that couldn't be translated.

Up till now, we are talking about the concept of exceptions in general, without projecting the concept on our architecture. In the next lines we show how exceptions are designed in our processor.

2.1.4.1 Exception Cases

Currently, these are all the events that cause our processor to raise an exception:

- **Interrupt**: Only if IEc flag of SR register is 1, if IRQ signal arrives to the processor, the processor sets ExcCode field of Cause register to 0, copies the address of instruction that is currently being executed into EPC register, then calls exception handler. We note that there is only one IRQ signal that is fed to the processor.

System designer should allow the IRQ signal to be shared by various I/O devices. Several techniques exist for sharing IRQ signals.

- **TLB miss**: When logical address fails to be translated (as descibed earlier), the processor sets ExcCode field of Cause register to 1, copies the logical address into BadVaddr register, and copies the address of the instruction that has caused the TLB miss into EPC register. The processor then calls the exception handler.

- **Reset signal**: Processor reset is also a form of exception, where the appropriate exception handler is called whenever this event happens. Reset doesn't cause the Cause register, EPC register, or any other registers to be modified. On RESET, software should re-initialize the processor and drives it to a known state.

It's important to note that when exception happens at some instruction, the processor makes sure that all previous instruction has done execution and their results has been commited successfuly before the exception handler is called. The instruction at which the exception has happened (the instruction whose address is copied to EPC) must not have any effect on the architectural state of the processor, so that when exception handler returns, the instruction can be re-executed without problems. We will later introduce the concept of precise exceptions.

2.1.4.2 Exception Vectors

Exception vector is the address where the exception handler is expected to reside. When exception happens, the processor performs a far jump to exception vector, starting from there the execution of the exception handler. Actually, one distinct exception vector could be assigned to every type of exception. In this case, there is an exception handler for every type of exception.

Exception vectors could be static or dynamic. For static exception vectors case, the vectors are predefined addresses that are hardwired in the processor. Static vectors can never change. Examples for architectures with static vectors include MIPS, 6502, and PIC architectures.

Dynamic vectors are set up by software. They could be stored inside one or more registers in the processor or could be stored in a table. The table will contains vectors for all possible exception types. Examples for architecture using dynamic vector mechanism include x86 and Zilog Z80. In some architectures, like 68K and PDP-11, the interrupting I/O device itself provides the processor with the interrupt vector when an interrupt is handled.

Since MIPS processors use static hard-wired exception vectors, our processor jumps to predefined addresses when exceptions happens. Figure 2.21 lists those predefined addresses and their associated exceptions.

Logical Address	Region	Physical Address	Exception Types
0xBFC00180	kseg1	0x1FC00180	All types of exception, except reset.
0xBFC00000	kseg1	0x1FC00000	Reset exceptions.

Figure 2.21: Excetion vectors for various exception types

We note that all exception handlers are located inside kseg1 memory region. This is important for reset vector to be located in kseg1 region as it is the only safe region to start the system from as explained before. Thus firmware location should be adjusted in memory (by system designer) such that first instruction of the firmware program matches reset vector.

The exception handler for other types of exception is also located in the uncached kseg1 region. This might be necessary during system initialization, but the OS later might want the exception handler to be called through kseg0 (instead of kseg1) so that it can be cached. A jump instruction could be inserted in the exception handler at kseg1 for that purpose.

2.1.4.3 Exception Protocol

Except for reset exceptions, the processor does the following when an exception happens:

1. Copy the address of the instruction that causes the exception into EPC register.
2. The 3-level stack inside SR register (explained before) is pushed and a 0 is inserted in IEc flag to disable interrupts.
3. Set the Cause register with appropriate ExcCode. If the exception is a TLB miss exception, the logical address is copied into BadVaddr register.
4. The processor transfers control to the exception handler.

The exception handler should go through these steps:

1. Store the state (context) of the interrupted program. EPC should also be stored.
2. Load Cause register and examine ExcCode field to determine the type of exception and call the appropriate handler.
3. Process the exception. This can alter the values of registers and change the state of the processor, that's why the context of the interrupted process is stored in first step.
4. Restore the state of the interrupted program.
5. Return from exception: this can be done by loading the value of the EPC register (that is stored in step 1) into one of the general purpose registers, then using JR instruction to jump back to the instruction referred to by EPC. RFE instruction should be inserted into the delay slot of the JR instruction so that the stack in the SR register is popped just before returning to the interrupted program (to enable interrupts again). This is the safest method to return from exception.

2.1.4.4 Precise Exceptions

We finally conclude our discussion with a note on the preciseness of exceptions. Due to pipelining, multiple exceptions may happen at the same time by various instructions in various stages. Furthermore, an exception might happen in an instruction where following instructions has already been executed and has changed the architectural state of the processor.

To handle these pipeline issues, exceptions need to be 'precise'. An exception is called precise if the saved processor state corresponds with the sequential model of the program execution, although the underlying hardware excutes instructions in parallel and out of order. In the case of precise exceptions, the OS knows well that all instructions before the instruction referred to by EPC have finished their execution successfully, and have already committed their results. The OS also knows that all instructions after EPC has never been executed and didn't effect the processor state. These two points is what we mean by saying that the saved processor state corresponds with the sequential model of execution.

Since our prcoessor implements precise exceptions, it guarantees the following:

- **Unambiguous proof of cause**: the values of EPC and Cause registers are exact and refer to the actual instruction that caused the exception and the actual type of exception. This wouldn't be the case on imprecise exceptions.
- **Exceptions are seen in instruction sequence**: as described above, precise exceptions guarantees that all instructions before EPC has finished their execution successfully. If several exceptions at several stages happen at the same time, the processor choose the earlier instruction of them to be put in EPC. When an exception happens at some stage of the pipeline, exception handler is not called until all instructions in the following stages finish execution and commit their results.
- **Subsequent instructions nullified**: all the stages before the stage that caused the exception are nullified and any change that happened by instructions in these stages is undone. This guarantees that all instructions after EPC can be restarted safely after exception handler returns.