EEL 4768
Computer Architecture

MIPS64

Outline

- Main Features
- Register Sets
- FPRs
- Memory Configuration
- Instruction Formats
- Arithmetic and Logic Operations
- Shift Operations
- Branch and Jump

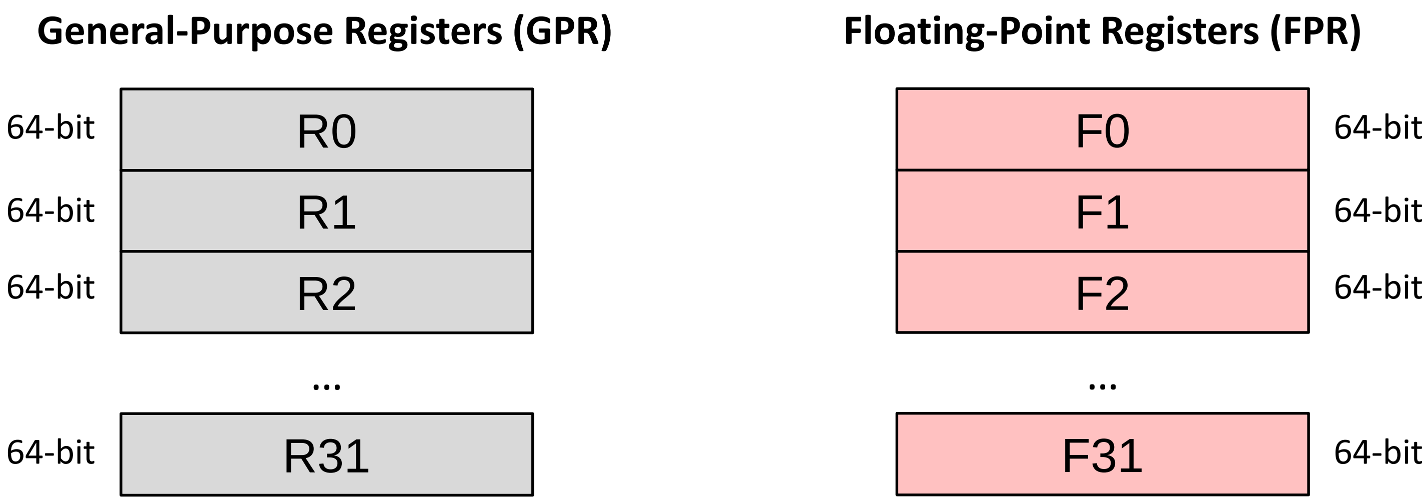
MIPS64: Main Features

- MIPS64 is the 64-bit version of MIPS32
- MIPS64 is a superset of MIPS32
  - MIPS32 instructions are included in MIPS64
  - Backward compatible: Can execute MIPS32 code
- Main features:
  - Load-store architecture w/ general-purpose registers
  - 32-bit instructions, similar in format to MIPS32
  - Supports a 64-bit memory address
  - Registers (integer and floating-point) are 64-bit
  - Condition Code Register (CCR) for conditional branches on floating-point values

MIPS64 is continuously updated every few years. The latest version is Release 6. These slides are based on App. A.9, which is an earlier release. Full MIPS instruction set is in App. K.

MIPS64: Register Sets

- Two sets of 64-bit registers:
  - General-Purpose Registers (GPR) hold **integer** values
  - Floating-Point Registers (FPR) hold **floating-point** values



- GPRs:
  - R0 is always equal to 0
  - Jump-and-link (jal) always links in register R31
- FPRs:
  - All are general-purpose and contain any value (in IEEE 754 format)

Data Types

- Integer data types
  - Bit 1-bit
  - Byte 8-bit
  - Halfword 16-bit
  - Word 32-bit
  - Doubleword 64-bit

- Floating-point data types
  - Single-precision 32-bit
  - Double-precision 64-bit

Integer vs. Floating-Point Operations

- Integer and floating-point operations are separate instructions, distinguished by the opcode

- DADD (Doubleword ADD) for integer addition

DADD R1, R2, R3

- ADD.D (Add Double-precision) for floating-point addition

ADD.D F1, F2, F3

Integer vs. Floating-Point Operations

Integer operations

Floating-point operations

General-Purpose Registers (GPR)

64-bit	R0
64-bit	R1
64-bit	R2
	…
64-bit	R31

Floating-Point Registers (FPR)

F0	64-bit
F1	64-bit
F2	64-bit
…	
F31	64-bit

Moving between GPRs and FPRs

also…

Converting data between integer and floating-point format

FPRs

- FPRs can hold:

One single-precision floating-point number

Single-Precision	000…….0
32-bit	32-bit

One double-precision floating-point number

Double-Precision
64-bit

Two single-precision numbers (single pairs)

Single-Precision	Single-Precision
32-bit	32-bit

FPRs: One Single-Precision

- Placed on the left side with padding of 32 bits of zeros
- Zeros don’t change the value!

- Format of an IEEE 754 floating-point number:

$$\mathbf{1.000010101 \times 2^{exponent}}$$

- The fields are:

Sign	Exponent	Significand

- The part “000010101” is the “Significand” field
- By placing the 0 on the right side, it becomes: <u>1.000010101</u>0000000…, which is the same value as the original number

FPRs: Single Pairs

- Two single-precision numbers in a 64-bit FPR
- Why would we put two numbers in one register?
- Register F1 has two single-precision numbers (A1, A2)
- Register F2 has two single-precision numbers (B1, B2)
- MIPS64 provides an instruction that makes two additions on F1 and F2
- This instruction will add (A1+B1) and (A2+B2) and stores the two results in F0

ADD.PS F0, F1, F2

- A suitable operation for processing matrices, for example:

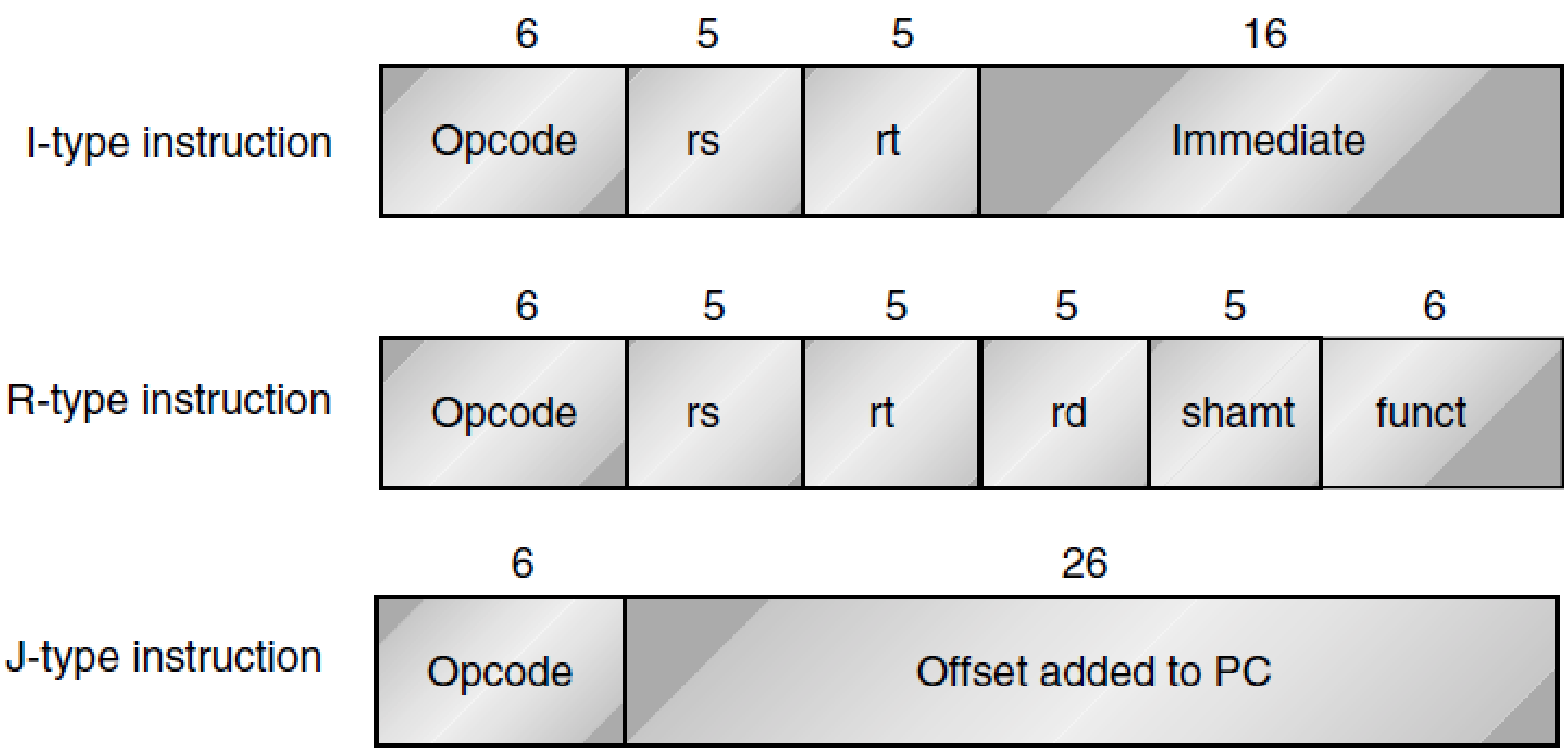
F0	(A1+B1)	(A2+B2)
F1	A1	A2
F2	B1	B2

Memory Configuration

- 64-bit memory addresses
- Byte addressable
  - Doubleword spans 8 addresses
  - Word spans 4 addresses
- The memory is aligned by default:
  - A byte data type can be at any address
  - A halfword’s (16-bit) address is a multiple of 2
  - A word’s (32-bit) address is a multiple of 4
  - A doubleword’s (64-bit) address is a multiple of 8
- Both the Big Endian scheme and the Little Endian coding (or word alignment) are supported:
  - A ‘mode bit’ in a configuration register selects between the two

Instruction Format

- The instructions are 32-bit, very similar to MIPS32
  - I-type: loads, stores and immediate instructions
  - R-type: register instructions
  - J-type: jump and jump-and-link



Integer Load and Store

Instruction	Syntax	Note
Load doubleword	LD R1, 80(R2)	64-bit integer in a GPR
Load word	LW R1, 40(R2)	32-bit integer in a GPR (sign-extend left 32 bits)
Load half	LH R1, 20(R2)	16-bit integer in a GPR (sign-extend left 48 bits)
Load byte	LB R1, 10(R2)	8-bit integer in a GPR (sign-extend left 56 bits)
Load word unsigned	LWU R1, 40(R2)	32-bit integer in a GPR (zero-extend left 32 bits)
Load half unsigned	LHU R1, 20(R2)	16-bit integer in a GPR (zero-extend left 48 bits)
Load byte unsigned	LBU R1, 10(R2)	8-bit integer in a GPR (zero-extend left 56 bits)
Store doubleword	SD R1, 80(R2)	64-bit in GPR stored in the memory
Store word	SW R1, 40(R2)	Rightmost 32-bit of GPR stored in the memory
Store half	SH R1, 20(R2)	Rightmost 16-bit of GPR stored in the memory
Store byte	SB R1, 10(R2)	Rightmost 8-bit of GPR stored in the memory

Floating-Point Load and Store

- The base register is a GPR since the address is an integer

Instruction	Syntax	Note
Load double-precision	L.D F0, 80(R2)	64-bit floating-point in an FPR
Load single-precision	L.S F0, 40(R2)	32-bit floating-point in the left half of the FPR
Store double-precision	S.D F0, 80(R2)	64-bit floating-point is stored in the memory
Store single-precision	S.S F0, 40(R2)	32-bit floating-point (left half of FPR) to memory

L.D F0, 80(R2)

Floating-point register

Integer register

Arithmetic Instructions

Instruction	Syntax	Note
Doubleword add	DADD R1, R2, R3	
Doubleword add immediate	DADDI R1, R0, 12	16-bit immediate
Doubleword add unsigned	DADDU R1, R2, R3	
Doubleword add unsigned immediate	DADDIU R1, R0, 1200000	The immediate is unsigned to support large values; if leftmost (bit#15) is 1, it's still a zero-extended positive
Doubleword sub	DSUB R1, R2, R3	
Doubleword sub unsigned	DSUBU R1, R2, R3	Integers interpreted as unsigned
Doubleword multiply	DMUL	
Doubleword multiply unsigned	DMULU	
Doubleword divide	DDIV	
Doubleword divide unsigned	DDIVU	

Handling of MIPS32

- The instructions below are still supported:

  ADD, ADDI, ADDU, ADDIU, SUB, SUBU

- They have different FUNCT fields
- For example, ADD and DADD use the opcode=0 but they use different function fields

- DADD will throw an exception if the 64-bit result overflows
- However, ADD will throw an exception if the 32-bit result overflows

Logic Operations

Instruction	Syntax	Note
AND operations	AND, ANDI	
OR operations	ORI, ORI	
XOR operations	XOR, XORI	
Set-on-less-than	SLT R1, R2, R3	(R1=1 if R2<R3), (Else R1=0)
Set-on-less-than immediate	SLTI R1, R2, 12	(R1=1 if R2<12), (Else R1=0)
Set-on-less-than unsigned	SLTU R1, R2, R3	Integers are interpreted as unsigned (leftmost bit is 1; number is positive)
Set-on-less-than unsigned immediate	SLTIU R1, R2, 1200000	

Shift

Instruction	**Syntax**	**Note**
Doubleword shift left logical	DSLL R1, R2, 4	Shift left from 0 bit to 31 bits
	DSLL32 R1, R2, 40	Shift left from 32 to 63 bits
Doubleword shift right logical	DSRL R1, R2, 4	Shift right by 4 bits
Doubleword shift right arithmetic	DSRA R1, R2, 4	Shift right arithmetic (preserve the leftmost bit) by 4 bits
Doubleword shift left logical variable	DSLLV R1, R2, R3	The shift amount is in register R3
Doubleword shift right logical variable	DSRLV R1, R2, R3	The shift amount is in register R3
Doubleword shift right arithmetic variable	DSRAV R1, R2, R3	The shift amount is in register R3

Shift

- The shift instructions use the ‘shift amount’ (shamt) field which is 5-bit
- Therefore, they can shift by up to 31 bits

- The DSLL32 is used to shift from 32 to 63 bits for the doubleword
- The ‘shamt’ field is incremented by 32

Handling 16 Bits

- This is the 'load upper immediate' (LUI) instruction

Load upper immediate	LUI R1, 0xAA34	Loads 16 bits in the 64-bit register R1… at which bit position?

- The position where the 16-bit immediate is loaded in the register

0x0000	0x0000	**0xAA34**	0x0000
16-bit	16-bit	16-bit	16-bit

- After the LUI, we may use 'ORI' to fill the rightmost 16 bits

Branch and Jump

Instruction	Syntax	Note
Jump	J Label	Unconditional jump
Jump-and-link	JAL Label	Jump and link the return address in R31
Jump-and-link register	JALR R1	Jump to address in R1; link in register R31
Branch-on-equal	BEQ R1, R2, Label	
Branch-on-not-equal	BNE R1, R2, Label	
Branch-on-equal zero	BEQZ R1, Label	Branch if R1=0
Branch-on-not-equal to zero	BNEZ R1, Label	Branch if R1 is not zero
Conditional move if zero	MOVZ R1, R2, R3	If R3=0, then do R1=R2

Floating-Point Compare for CCR

Instruction	**Syntax**	**Note**
Compare equal	C.EQ.S F0, F1	F0 = F1 ?
Compare not equal	C.NE.S F0, F1	F0 != F1 ?
Compare less than	C.LT.S F0, F1	F0 < F1 ?
Compare less or equal	C.LE.S F0, F1	F0 <= F1 ?
Compare greater than	C.GT.S F0, F1	F0 > F1 ?
Compare greater or equal	C.GE.S F0, F1	F0 >= F1 ?
Same as above; but for double-precision	C.EQ.D F0, F1	
	C.NE.D F0, F1	
	C.LT.D F0, F1	
	C.LE.D F0, F1	
	C.GT.D F0, F1	
	C.GE.D F0, F1	

Floating-Point Compare and Branch

- Where is the result of the comparison?

C.EQ.S F0, F1

- It's stored in a special bit in "coprocessor1"
- After the comparison, we can use the instructions below to do a branch based on the comparison

Branch if coprocessor1 bit is true	BC1T Label	Branch if the result of the comparison is true
Branch is coprocessor1 bit is false	BC1F Label	Branch if the result of the comparison is false

- There should not be any instruction in between, e.g.:

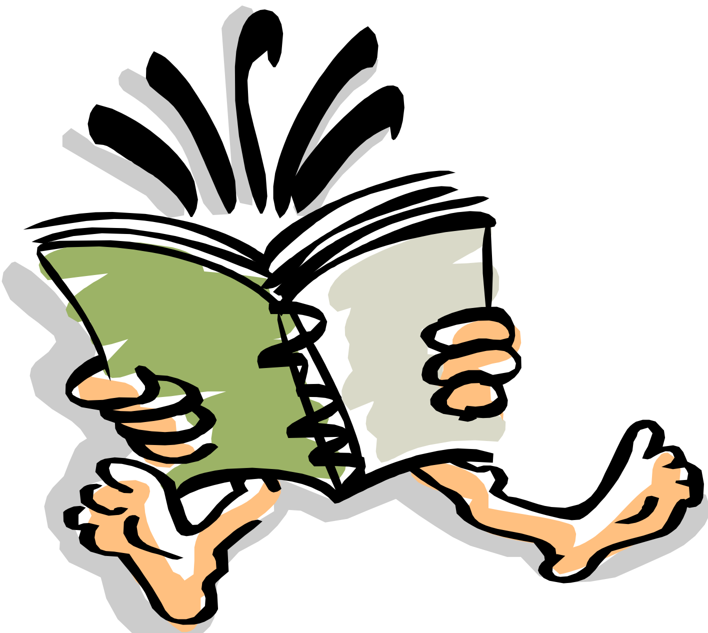
C.EQ.S F0, F1

BC1T Label # branch if F0=F1

Summary

- MIPS64 is backwards compatible
- Still, have to handle <64-bit operands carefully
- FPRs and GPRs
- Allows Condition Code Register (CCR) style branching

Readings



- H&P CA
  - App A
  - App K