EEL 4768: Computer Architecture

MIPS64

Instructor: Zakhia (Zak) Abichar

Department of Electrical Engineering and Computer Science
University of Central Florida

"Computer Architecture, A Quantitative Approach", **Appendix A.9**

MIPS64

- MIPS64 is the 64-bit version of MIPS32
- It's backward compatible with MIPS32
- A MIPS32 binary code would execute on a MIPS64 CPU
- MIPS64 is a superset of MIPS32
- All of MIPS32 instructions are included in the MIPS64 instruction set

MIPS64

- MIPS64 has many similarities with MIPS32
- It's a load-store architecture that relies on the general-purpose registers

The main features of MIPS64 are:

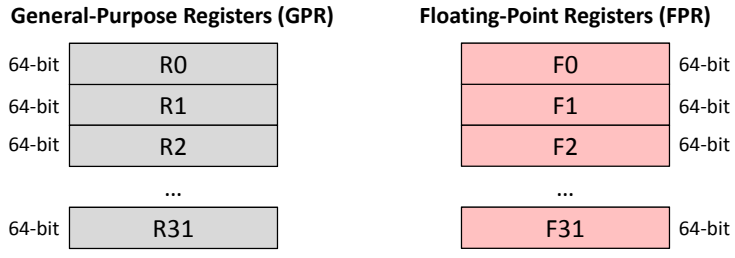
- It uses 32-bit instructions, similar in format to MIPS32
- It supports a 64-bit memory address
- The registers (integer and floating-point) are 64-bit
- It uses the 'condition code register' approach for conditional branches on floating-point values

MIPS64 has been published in multiple versions and is continuously updated every few years. The latest version is Release 6. These slides are based on Appendix A.9 of the book, which is an earlier release.

3

MIPS64

- MIPS64 has two sets of registers
- The General-Purpose Registers (GPR) hold integer values
- The Floating-Point Registers (FPR) hold **floating-point** values
- All the registers are 64-bit



4

MIPS64

- Integer operations and floating-point operations are distinguished via the opcode
- There are separate instructions for each type

- DADD (Doubleword ADD) for integer operation

DADD R1, R2, R3

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

ADD.D F1, F2, F3

5

MIPS64

- This is the terminology for the data types:

Integer data types

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

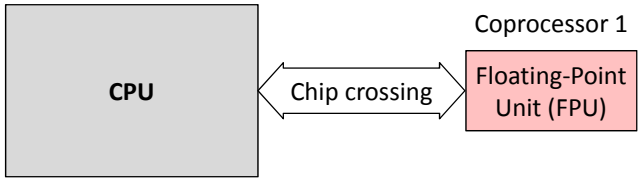
Floating-point data types

- Single-precision 32-bit
- Double-precision 64-bit

6

MIPS64

- Historically, the Floating-Point Unit (FPU) used to be its own chip
- It was called the **'Coprocessor 1'** or **'c1'**
- An external connection, the **'chip crossing'** connects the FPU to the CPU
- The FPR registers were located on the FPU

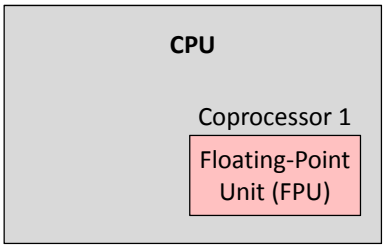


A coprocessor is a small unit that helps the processor by providing special instructions; coprocessor0 usually deals with memory access and exception handling. Another coprocessor may provide parallel instructions for matrix operations.

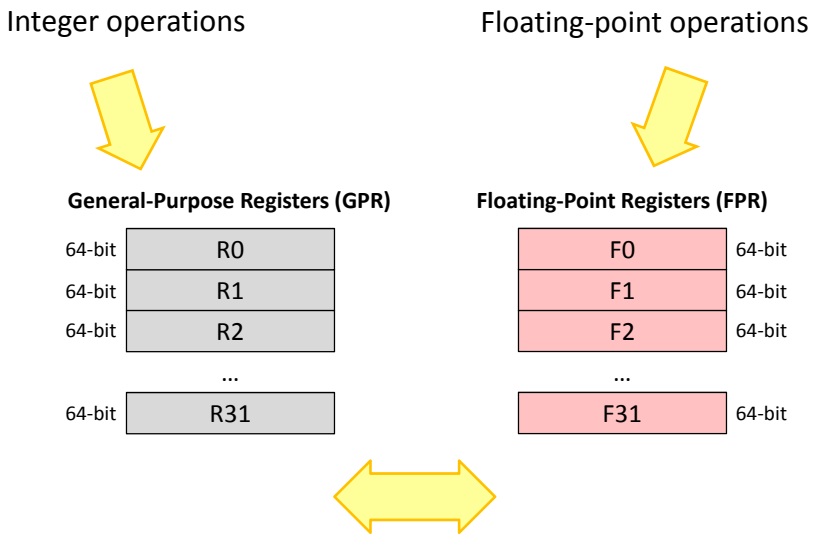
7

MIPS64

- Nowadays, the CPU has many transistors that the FPU resides on the CPU
- However, the FPU is still called coprocessor1



8



Moving between GPRs and FPRs
also…
Converting data between integer and floating-point format

9

MIPS64

Integer registers

- R0 is always equal to 0
- Jump-and-link (jal) always links in register R31

Floating-point registers

- All the registers F0 to F31 are general-purpose
- They may contain any value (in IEEE 754 format)

10

MIPS64

- A floating-point register can be used to hold one of the choices below:

(1) One single-precision (32-bit) number
Since the register is 64-bit, one half of the register is unused

(2) One double-precision (64-bit) number

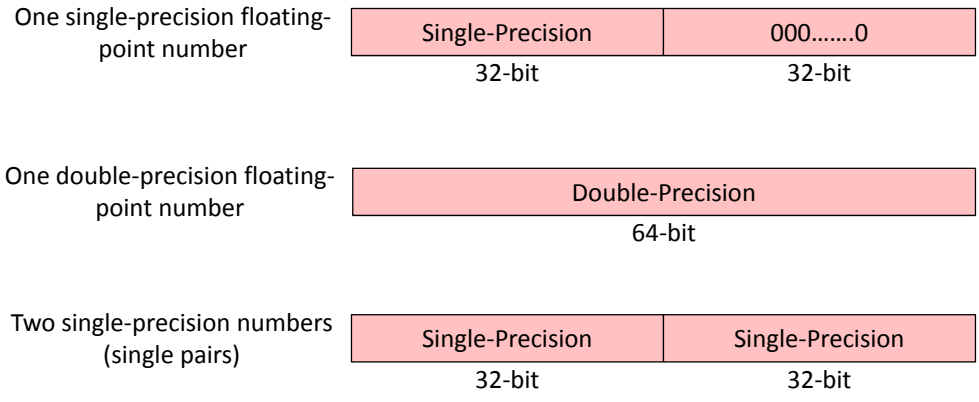
(3) Two single-precision numbers side-by-side, a configuration called **'single-pairs'**

- Shown in figure (next slide)

11

MIPS64

- Notice, in the first case, the single-precision number is placed in the left side of the 64-bit register



12

MIPS64

- A single-precision number is placed in the left side of an FRP with the remaining right 32 bits filled with zero
- This makes it seem like adding zeros on the right side of the decimal point (which don't change the number)

- This is the format of a IEEE 754 Standard floating-point number

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

- The fields are:

Sign	Exponent	Significand

- The part "000010101" is the "Significand" field
- By placing the 0 on the right side, it becomes: 1.0000101010000000…, which is the same value as the original number

13

MIPS64

- The 'single pairs' configuration places two single-precision numbers in a 32-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

14

MIPS64

Memory configuration

- The memory in MIPS64 address is 64-bit
- The memory is byte addressable; therefore a 64-bit number (8-byte) occupies 8 addresses in the memory
- For example, placing a word at address 0 will occupied the addresses 0 through 7

- The memory is aligned (misaligned memory is also supported)
- 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

15

MIPS64

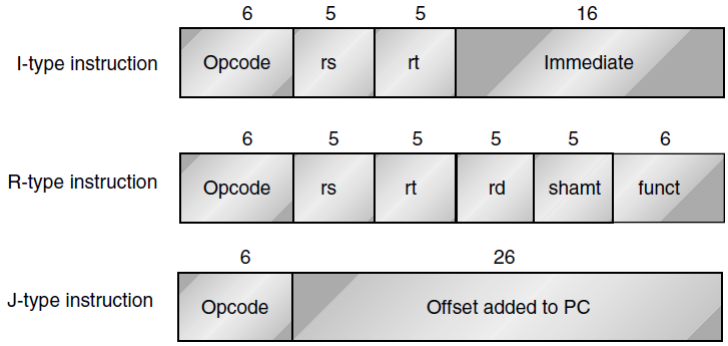
- Both the Big Endian scheme and the Little Endian scheme are supported in MIPS64
- A 'mode bit' in a configuration register selects between the two schemes

16

MIPS64

Instruction format

- The instructions are 32-bit and they're fixed size
- The format is similar to MIPS32
- The I-type is used for loads, stores and immediate instructions
- The R-type is used for register instructions
- The J-type is used for jump and jump-and-link



17

These are the load and store instructions for integer data types

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

18

MIPS64

- These are the load and store instruction that operate on the FPR registers
- Notice that the base register is an integer register 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

19

MIPS64

- These are the 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 immediate values; if leftmost (bit#15) is 1, it's still positive an zero-extended
Doubleword sub	DSUB R1, R2, R3	
Doubleword sub unsigned	DSUBU R1, R2, R3	Integers interpreted as unsigned

20

MIPS64

- The instructions below are still supported:

ADD, ADDI, ADDU, ADDIU, SUB, SUBU

- They have different FUNC 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

21

MIPS64

- These are the 'slt' (set-on-less-than)

Instruction	Syntax	Note
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	

- There is one version of 'SLT' instructions that work on the 32-bit and 64-bit operands

22

MIPS64

- These are the logic instructions

Instruction	Syntax	Note
Doubleword multiply	DMUL	
Doubleword multiply unsigned	DMULU	
Doubleword divide	DDIV	
Doubleword divide unsigned	DDIVU	
AND operations	AND, ANDI	
OR operations	ORI, ORI	
XOR operations	XOR, XORI	

- MUL is used for 32-bit operands

23

MIPS64

- Difference between MUL and MULT
- MUL

rs (32-bit) x rt (32-bit) → rd (32-bit)

- MULT

rs (32-bit) x rt (32-bit) → HI – LO (64-bit)

24

MIPS64

- These are the shift instructions

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

25

MIPS64

- 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

26

MIPS64

- 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 figure below shows the position where the 16-bit immediate is loaded in the register
- Presumably, after the 'lui', we can use 'ORI' to fill the rightmost 16 bits of the register

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

27

MIPS64

- These are the branch and jump instructions

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

28

- The compare instruction is used to compare floating-point register

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	

29

MIPS64

- Let's say we did the instruction below to see if F0 is equal to F1
- Where is the result of the comparison?

C.EQ.S F0, F1

- It's stored in a special bit in coprocessor1
- After doing 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

30

MIPS64

- Therefore, this is a sequence that can be used to branch based on floating-point registers

```
C.EQ.S      F0, F1
BC1T        Label       # branch if F0=F1
```

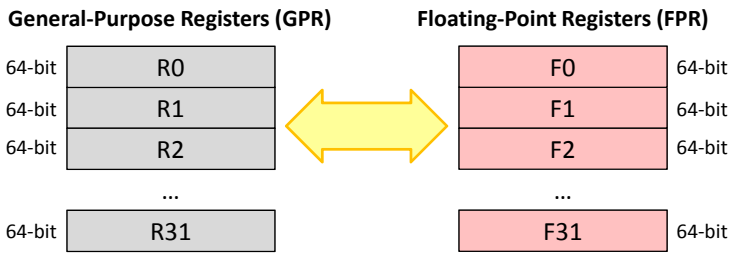
- There is feedback from the compare to the branch
- Compare sets the coprocessor1 bit and the branch reads this bit
- Therefore, there should not be any instruction between them that may alter this bit

31

MIPS64

- The instructions below move the data between GPRs and FPRs
- The two instructions below copy the data bit-by-bit; they don't convert between integer and IEEE 754 format
- Other conversion instructions will do this

Move from coprocessor1	MFC1 R1, F1	FPR copied into GPR; format is not converted
Move to coprocessor1	MTC1 F1, R1	GPR copied into FPR; format is not converted



32

MIPS64

- These are the data types that are used by the ‘convert (CVT) instruction

- 32-bit integer — word (W)
- 64-bit integer — long word (L)
- 32-bit floating-point — single-precision (S)
- 64-bit floating-point — double-precision (D)

33

MIPS64

- This is the convert (CVT) instruction

Instruction	Syntax	Note
32-bit integer (W) to double-precision(D)	CVT.D.W F0, F1	
64-bit integer (L) to double-precision (D)	CVT.D.L F0, F1	
32-bit single-precision(S) to double-precision (D)	CVT.D.S F0, F1	
32-bit integer (W) to single-precision (S)	CVT.S.W F0, F1	
64-bit integer (L) to single-precision (S)	CVT.S.L F0, F1	
64-bit double-precision (D) to single-precision (S)	CVT.S.D F0, F1	

34

MIPS64

- These CVT variants convert to integer format

Instruction	Syntax	Note
	CVT.L.S F0, F1	Single-precision (S) to 64-bit integer (L)
	CVT.L.D F0, F1	Double-precision (D) to 64-bit integer (L)
	CVT.W.S F0, F1	Single-precision (S) to 32-bit integer (W)
	CVT.W.D F0, F1	Double-precision (D) to 32-bit integer (W)

- There's no: CVT.L.W
- A word (32-bit) is already stored in a 64-bit register; there's no need to convert it to 64-bit

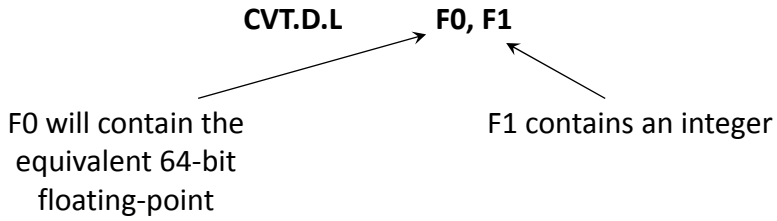
- There's no: CVT.W.L
- We don't need to convert 64-bit to 32-bit since the register is 64-bit

35

MIPS64

- The convert instruction operates on FPRs only

- Even when the data is integer, the CVT takes FPRs only
- Therefore, the integer is copied from GPR to FPR and then converted to floatin-point



36

MIPS64

- To convert this code into MIPS64… we'll do an integer addition

```
int a;                  // in R1 32-bit integer
float f;                // in F1 32-bit floating-point

a = a + (int)f;         // integer addition
```

- The variable 'f' should be converted to integer
- The conversion should happen in the FPR before moving 'f' into a GPR

```
CVT.W.S     F31, F1     # convert from single-precision to 32-bit integer
MFC1        R30, F31
ADD         R1, R1, R30
```

37

MIPS64

- To convert this code into MIPS64… we'll do a floating-point addition

```
int a;                  // in R1 32-bit integer
float f;                // in F1 32-bit floating-point

f = f + (float)a;       // floating-point addition
```

- The variable 'a' should be converted to float
- The conversion should happen in the FPR; therefore, we should start by moving into the FPR followed by the conversion

```
MTC1        F31, R1
CVT.S.W     F31, F31    # convert from 32-bit integer to single-precision
ADD.S       F1, F1, F31
```

38

MIPS64

- The move instructions below are used to copy one FPR into another
- In the GPRs, we can use register R0 to copy one register to another (such as copying R2 into R1 via: DADD R1, R2, R0)
- However, in the FPR, we don't have the value zero readily available; that's why the move instructions are provided

Instruction	Syntax	Note
Move single-precision	MOV.S F0, F1	F0 = F1
Move double-precision	MOV.D F0, F1	F0 = F1

- The instruction (.S) or (.D) should correspond to the data type in the FPR

39

These are arithmetic instructions on the floating-point registers

Instruction	Syntax	Note
Add double-precision	ADD.D	
Add single-precision	ADD.S	
Add single pairs	ADD.PS	
Subtract double-precision	SUB.D	
Subtract single-precision	SUB.S	
Subtract single pairs	SUB.PS	
Multiply double-precision	MUL.D	
Multiply single-precision	MUL.S	
Multiply single pairs	MUL.PS	
Divide double-precision	DIV.D	
Divide single-precision	DIV.S	
Divide single pairs	DIV.PS	

40

MIPS64

- The assembler provides the use of data directives to declare variables in the code
- The directives below differentiate between the data types

.word	64-bit integer
.word32	32-bit integer
.word16	16-bit integer
.byte	8-bit integer
.float	32-bit floating-point
.double	64-bit floating-point

41

- Using the assembler data directives

```
.data           # Data segment
ch:     .byte       1
sh:     .word16     2
n:      .word32     3
x:      .word       4
f:      .float      5.6
y:      .double     7.8

.text           # Text segment
LA      R30, ch     # load address of 'ch'
LB      R1, 0(R30)  # load 'ch' in R1 using LB
LA      R30, sh
LH      R2, 0(R30)  # load 'sh' in R2 using LH
LA      R30, n
LW      R3, 0(R30)  # load 'n' in R3 using LW
LA      R30, x
LD      R4, 0(R30)  # load 'x' in R4 using LD
LA      R30, f
L.S     F0, 0(R30)  # load 'f' in F0 using L.S
LA      R30, y
L.D     F1, 0(R30)  # load 'y' in F1 using L.D
...
```

```
char ch=1;          // 8-bit int
short int sh=2;     // 16-bit int
int n=3;            // 32-bit int
long int x=4;       // 64-bit int
float f=5.6;        // 32-bit FP
double y=7.8;       // 64-bit FP
...
```

42

MIPS64

- Write a MIPS64 code that evaluates this inequality

$$|a^2 - b| < epsilon$$

- The variables ‘a’, ‘b’ and ‘epsilon’ are of type **‘float’**

43

MIPS64

```
.data                       # declaring the data in the program's memory
a:      .float 0.1
b:      .float 0.01
e:      .float 1.0e-7

.text
LA      R1, a               # next 3 instructions load the address of
LA      R2, b               # the variables
LA      R3, e
L.S     F0, 0(R1)           # F0 <- a
L.S     F1, 0(R2)           # F1 <- b
L.S     F2, 0(R3)           # F2 <- epsilon
MUL.S   F0, F0, F0
SUB.S   F3, F0, F1
ABS.S   F3, F3              # computes the absolute value (single-precision)
C.LT.S  F3, F2
BC1F    not_quite
…
not_quite
```

44

MIPS64

- What does this code do?

```
cvt.w.s    F31, F0
mfc1       R2, F31
add        R3, R1, R2
```

45

MIPS64

- This is the code with comments

```
cvt.w.s   F31, F0      # convert from single-precision to 32-bit integer
mfc1      R2, F31      # copy the integer to register R2
add       R3, R1, R2   # add R2 to R1
```

- This code converts a floating-point value in an FPR to integer type, copies it into an integer register and adds it to another integer register

46

MIPS64

- Load the 64-bit number 0x11223344 AABBCCDD in register R1

```
long int n = 4;
...
n = 0x11223344AABBCCDD;
```

```
.data
n:      .word   4               #initial value

.text
...
LUI     R1, 0x1122              # R1: 0000 0000 1122 0000
ORI     R1, R1, 0x3344          # R1: 0000 0000 1122 3344
DSLL32  R1, R1, 32              # R1: 1122 3344 0000 0000
LUI     R2, 0xAABB              # R2: 0000 0000 AABB 0000
ORI     R2, R2, 0xCCDD          # R2: 0000 0000 AABB CCDD
OR      R1, R1, R2              # R1: 1122 3344 AABB CCDD

LA      R30, n
SD      R1, 0(R30)              # store the value in 'n' in the memory
```

47

MIPS64

- This is another way to do this code
- If a constant is used often, we can store it in the memory with the program instead of computing this value with 'lui' and 'ori'

```
long int n=4;
...
n = 0x11223344AABBCCDD;
```

```
.data
n:      .word   4
const:  .word   0x11223344AABBCCDD

.text
...
LA      R30, const
LD      R1, 0(R30)      # contains the 64-bit value

LA      R30, n
SD      R1, 0(R30)      # store the 64-bit constant in 'n' at the memory
```

48

MIPS64

- The code below converts a Fahrenheit temperature reading into Celsius

```
double cel, fah;
...
cel = (fah – 32) *5/9;
```

- The division 5/9 has to be done as a floating-point division
- If it were done as an integer division, it yields zero

- How can we load the constants 5, 9 and 32 as floating-point values?
- We can't do ADDI with the floating-point
- We can either load them as constants with the program (using .double data directive)
- Or we can load the '5' and '9' as integers (with ADDI), then convert them to floating-point using 'CVT'

49

```
double cel, fah;
...
cel = (fah – 32) *5/9;
```

```
.data
cel:      .double     ...
fah:      .double     ...
const5:   .double     5       # 5 stored in IEEE 754 format
const9:   .double     9       # 9 stored in IEEE 754 format
const32:  .double     32      # 32 stored in IEEE 754 format

.text
LA        R30, fah
L.D       F1, 0(R30)          # F1 <- fah
LA        R30, const5
L.D       F2, 0(R30)          # F2 <- 5
LA        R30, const9
L.D       F3, 0(R30)          # F3 <- 9
LA        R30, const32
L.D       F3, 0(R30)          # F4 <- 32

SUB.D     F0, F1, F4          # doing (fah-32)
MUL.D     F0, F0, F2          # multiply by 5
DIV.D     F0, F0, F3          # divide by 9

LA        R30, cel
S.D       F0, 0(R30)
```

We're using a lot of 'LA' instructions. We'd better reference the variables with respect to a Global Pointer (as in $gp in MIPS32)

50

MIPS64

- In the next slide, we'll do the same program but we'll load the constants 5, 9 and 32 as integers using DADDI then convert them to floating-point

51

```
double cel, fah;
...
cel = (fah – 32) *5/9;
```

```
.data
cel:    .double     ...
fah:    .double     ...

.text
DADDI       R1, R0, 5
DADDI       R2, R0, 9
DADDI       R3, R0, 32
MTC1        F1, R1
MTC1        F2, R2
MTC1        F3, R3
CVT.D.L     F1, F1          # constant 5 in floating-point
CVT.D.L     F2, F2          # constant 9 in floating-point
CVT.D.L     F3, F3          # constant 32 in floating-point

LA      R30, fah
L.D     F0, 0(R30)
SUB.D   F0, F0, F3          # doing (fah-32)
MUL.D   F0, F0, F1          # multiply by 5
DIV.D   F0, F0, F2          # divide by 9

LA      R30, cel
S.D     F0, 0(R30)
```

52

MIPS64

- Finally, we can rely on the pseudo-instructions to load a floating-point constant
- The assembler will store the constant as part of the program (like our previous code)

Instruction	Syntax	Note
Load immediate single-precision	LI.S F0, 2.3	
Load immediate double-precision	LI.D F0, 3.445	

53

double cel, fah;
...
cel = (fah – 32) *5/9;

```
.data
cel:    .double     ...
fah:    .double     ...

.text
LA      R30, fah
L.D     F0, 0(R30)          # fah loaded in F0

LI.D    F1, 5               # pseudo-instruction
LI.D    F2, 9
LI.D    F3, 32

SUB.D   F0, F0, F3          # subtract 32
MUL.D   F0, F0, F1          # multiply by 5
DIV.D   F0, F0, F2          # divide by 9

LA      R30, cel
S.D     F0, 0(R30)
```

54

MIPS64

- Translate the C code below into MIPS64 assembly

```
long int a, b, c;          // 64-bit integers
float average;             // 32-bit float
average = (float) (a+b+c)/3;
```

- The code is doing a floating-point division

- a @ 1000
- b @ 1008
- c @ 1016
- average @ 2000

55

```
long int a, b, c;
float average;
average = (float) (a+b+c)/3;
```

```
.data:
a:      .word   ...     @1000
b:      .word   ...     @1008
c:      .word   ...     @1016
avg:    .float  ...     @2000

.text:
LD      R1, 1000(R0)            # load a
LD      R2, 1008(R0)            # load b
LD      R3, 1016(R0)            # load c
DADD    R4, R1, R2
DADD    R4, R4, R3
MTC1    F0, R4                  # move the sum to an FPR
CVT.S.L F0, F0                  # convert the sum to a single-precision number

LI.D    F1, 3
DIV.S   F2, F0, F1
S.S     F2, 2000(R0)
```

56

MIPS64

- Translate the C code below into MIPS64 assembly

```
double a, b, c, average;    // 64-bit floats
...
average = (a+b+c)/3;
```

- a @ 1000
- b @ 1008
- c @ 1016
- avg @ 2000

57

```
double a, b, c, average
average = (a+b+c)/3;
```

```
.data:
a:      .word   ...     @1000
b:      .word   ...     @1008
c:      .word   ...     @1016
avg:    .float  ...     @2000

.text:
L.D     F0, 1000(R0)            # load a
L.D     F1, 1008(R0)            # load b
L.D     F2, 1016(R0)            # load c
ADD.D   F3, F0, F1
ADD.D   F3, F3, F2

LI.D    F4, 3                   # F4 <- 3.0

DIV.D   F3, F3, F4
S.D     F3, 2000(R0)
```

58

MIPS64

- Translate the C code below into MIPS64 assembly

```
long int A, B, F;              // 64-bit integer
…
if (A==0 && B==25)
        F = A + B;
```

- A @ 800
- B @ 808
- F @ 816

59

```
.data
A:      .word   ...     @800
B:      .word   ...     @808
F:      .word   ...     @816

.text
LD      R1, 800(R0)             # R1 <- A
BNE     R1, R0, Exit            # if A!=0, exit

LD      R2, 808(R0)             # R2 <- B
DADDI   R3, R0, 25
BNE     R2, R3, Exit            # R3 <- 25

DADD    R4, R1, R2
SD      R4, 816(R0)             # store the result in 'F' in the memory

Exit:
```

```
long int A, B, F;              // 64-bit integer
…
if (A==0 && B==25)
        F = A + B;
```

60

MIPS64

- Translate the C code below into MIPS64 assembly
- It's the same code as the previous one, except that the variables here are double-precision floating-point

```
double A, B, F;        // 64-bit floating-point
…
if (A==0 && B==25)
        F = A + B;
```

- How do we compare to zero?
- Zero as floating-point is not readily available; so we have to load it like we will do for 25

- A @ 800
- B @ 808
- F @ 816

61

```
double A, B, F;        // 64-bit floating-point
…
if (A==0 && B==25)
        F = A + B;
```

```
.data
A:      .double  ...      @800
B:      .double  ...      @808
F:      .double  ...      @816

.text
LI.D    F0, 0                   # F0 <- 0
L.D     F1, 800(R0)             # F1 <- A
C.EQ.D          F0, F1
BC1F    Exit

LI.D    F2, 25                  # F2 <- 25
L.D     F3, 808(R0)             # F3 <- B
C.EQ.D          F2, F3
BC1F    Exit

ADD.D   F4, F1, F3              # A+B
S.D     F4, 816(R0)             # store the result in 'F' in the memory

Exit:
```

62

MIPS64

- Translate the C code below into MIPS64 assembly

```
int i, A;
…
for (i=0; i<10; i++)
        A = A + 15;
```

- i @ 800
- A @ 808

63

```
.data
i:      .word32     ...     @800
A:      .word32     ...     @808

.text
        ADD     R1, R0, R0          # i <- 0
        ADDI    R2, R0, 10          # R2 <- 10
        LW      R3, 800(R0)         # R3 <- A

Loop:   BEQ     R1, R2, Exit
        ADDI    R3, R3, 15
        ADDI    R1, R1, 1
        J       Loop

Exit:   SW      R3, 808(R0)         # store A in memory
        SW      R1, 800(R0)         # store i in memory
```

```
int i, A;
…
for (i=0; i<10; i++)
        A = A + 15;
```

64

MIPS64

- Translate the C code below into MIPS64 assembly

```
long int A[] = {...};          // array of 64-bit integers
long int B[] = {...};          // array of 64-bit integers
long int C, i;                 // 64-bit integers
...
for (i=0; i<=100; i++)
        A[i] = B[i] + C;
```

- Use these addresses:
  - C @1000
  - i @1200
  - A @2400
  - B @4800

65

```
DADD   R1, R0, R0        # The variable 'i' is set to 0
LD     R2, 1000(R0)      # load C once outside of the loop
Loop:
DSLL   R3, R1, 3         # Compute i*8
DADDI  R4, R3, 2400      # This is the address of A[i] (it's: 2400+8*i)
DADDI  R5, R3, 4800      # This is the address of B[i] (it's: 4800+8*i)
LD     R6, 0(R5)         # load B[i]
DADD   R6, R6, R2        # B[i] + C
SD     R6, 0(R4)         # store the result in A[i]

DADDI  R1, R1, 1         # increment i
DADDI  R7, R1, -101      # has the counter reached 101?
BNEZ   R7, loop          # if not 101 then repeat
SD     R1, 1200(R0)      # store the counter 'i' in the memory
```

66

MIPS64

- Translate this code that finds the maximum float value in the array

```
double arr[40] = {2.3, 4.3, ...};      // 64-bit floating-point
double max;                            // 64-bit floating-point
int i;                                 // 32-bit integer
max = arr[0];
for(i=0; i<40; i++) {
        if(arr[i] > max)
                max = arr[i];
}
```

- These are the addresses:

  i @1000

  max @1008

  arr @2000

67

```
DADDI   R1, R0, 2000    # point at the array
DADDI   R2, R1, 320     # end of array (40 elements x 8 bytes)

L.D     F0, 0(R1)       # F0 is max; initialized to first array location

Loop:
BEQ     R1, R2, Exit
L.D     F1, 0(R1)       # F1 <- array data
C.GT.D  F1, F0          # is new data larger than max; F1 > F0 ?
BC1F    Skip            # if not, don't change anything
MOV.D   F0, F1          # if yes, F0 is set to F1
Skip:
DADDI   R1, R1, 8
J       Loop

S.D     F0, 1008(R0)    # max is set to the maximum value found

ADDI    R3, R0, 40
SW      R3, 1000(R0)    # when the code finishes, i=40
```

68

MIPS64

- Translate the program into MIPS64 assembly code

```
float A1, A2;       // 32-bit floating-point
float B1, B2;
float C1, C2;
...
C1 = A1+B1;
C2 = A2+B2;
```

- This is the memory layout

Memory		
80:	A1	Single-precision (32-bit)
84:	A2	Single-precision (32-bit)
…		
160:	B1	Single-precision (32-bit)
164:	B2	Single-precision (32-bit)
…		
800:	C1	
804:	C2	

69

MIPS64

```
L.S     F0, 80(R0)          # F0 <- A1
L.S     F1, 84(R0)          # F1 <- A2
L.S     F2, 160(R0)         # F2 <- B1
L.S     F3, 164(R0)         # F3 <- B2

ADD.S   F4, F0, F2          # F4 <- A1 + B1
ADD.S   F5, F1, F3          # F5 <- A2 + B2

S.S     F4, 800(R0)         # C1 <- (A1+B1)
S.S     F5, 804(R0)         # C2 <- (A2+B2)
```

70

MIPS64

- This is another way to do the code using the 'single pairs'

```
L.D       F1, 80(R0)          # F0 <- (A1, A2)
L.D       F2, 160(R0)         # F2 <- (B1, B2)

ADD.PS    F0, F1, F2

S.D       F0, 800(R0)
```

- The advantage of using 'single pairs' is reducing the number of instructions fetched from the memory (4 vs 8 in previous slide)

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

Using single pairs, load A1 and A2 in F1; and load B1 and B2 in F2; do one single pairs addition

71