EEL 4768
Computer Architecture

MIPS64 Examples

Outline

- Conversions
- Floating-Point Arithmetic
- Examples

Transfer Between FPRs

- In 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

- The move instructions below are used to copy one FPR into another

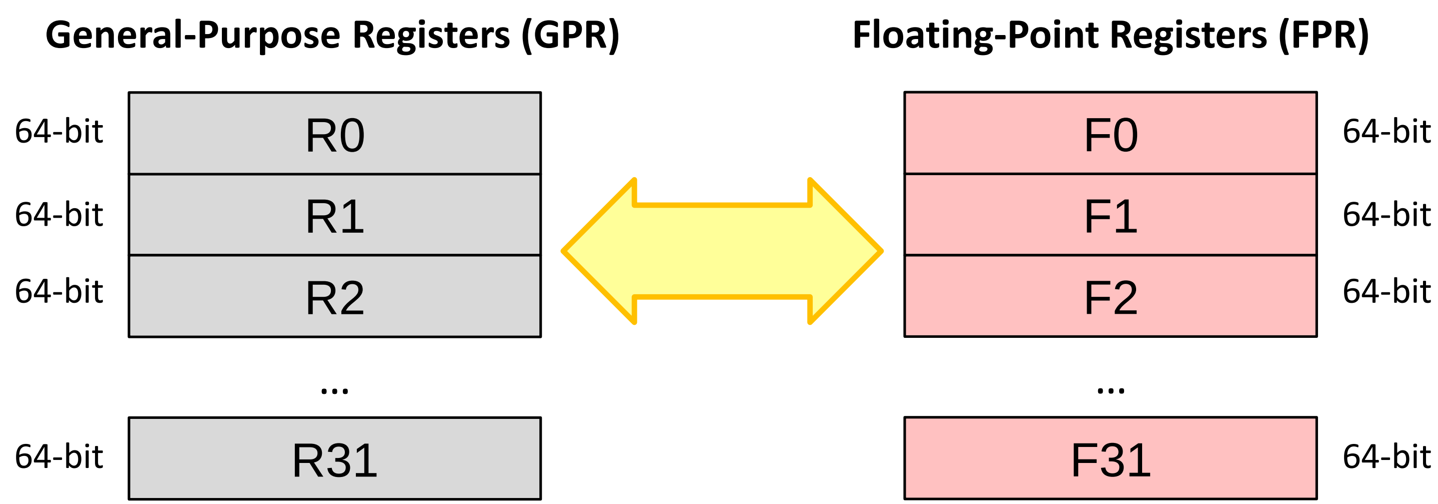
Instruction	Syntax	Note
Move single-precision	MOV.S F0, F1	F0 = F1
Move double-precision	MOV.D F0, F1	F0 = F1

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

Transfers Between FPRs and GPRs

- The two instructions below copy the data bit-by-bit; they don't convert between integer and IEEE 754 format
- Conversion instructions are needed to convert

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



Conversions

Instruction	Syntax	Note
	CVT.D.W F0, F1	32-bit integer (W) to double-precision(D)
	CVT.D.L F0, F1	64-bit integer (L) to double-precision (D)
	CVT.D.S F0, F1	32-bit single-precision(S) to double-precision (D)
	CVT.S.W F0, F1	32-bit integer (W) to single-precision (S)
	CVT.S.L F0, F1	64-bit integer (L) to single-precision (S)
	CVT.S.D F0, F1	64-bit double-precision (D) to single-precision (S)
	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 or CVT.W.L
  - L or W, the register they are stored in is already a 64-bit register

Conversions

- 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 floating-point

CVT.D.L F0, F1

F0 will contain the equivalent 64-bit floating-point

F1 contains an integer

Conversion Examples

- 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
```

Conversion Examples

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

Floating-Point Arithmetic

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	

Assembler Data Directives

- 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

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
...
```

Examples

- Write a MIPS64 code that evaluates this inequality:

$$|a^2 - b| < \texttt{epsilon}$$

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

Examples

```
.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
```

Examples

- What does this code do?:

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

Examples

- 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

Examples: Load a 64-bit Number

- Load the 64-bit number 0x11223344 AABBCCDD to R1
- We can do SLL and ORIs

```
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: 1111 1111 AABB 0000
ORI       R2, R2, 0xCCDD              # R2: 1111 1111 AABB CCDD
DSLL32 R2, R2, 32
DSRL32 R2, R2, 32
OR        R1, R1, R2                  # R1: 1122 3344 AABB CCDD

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

Examples: Load a 64-bit Integer Value

- 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
```

What’s the catch? Why bother with lui and ori?

Examples: Loading a Floating-Point Value

- 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’

Examples: Loading a Floating-Point Value

```
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       F4, 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)

Examples: Loading a Floating-Point Value

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

.text
DADDI          R1, R0, 5
DADDI          R2, R0, 9
DADDI          R3, R0, 32
MTC1           R1, F1
MTC1           R2, F2
MTC1           R3, F3
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)
```

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

Examples: Loading a Floating-Point Value

- 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	

Examples: Loading a Floating-Point Value

```
.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)
```

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

Examples

- 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

Examples

```
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    R4, F0                  # move the sum to an FPR
CVT.S.L F0, F0                  # convert the sum to a single-precision number

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

Examples

- 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

Examples

double a, b, c, average

average = (a+b+c)/3;

```
.data:
a:      .double  ...      @1000
b:      .double ...       @1008
c:      .double ...       @1016
avg:    .double  ...      @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)
```

Examples

- 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

Examples

```
.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;
```

Examples

- 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

Examples

```
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:
```

Examples: For Loop

- Translate the C code below into MIPS64 assembly

```
int i, A;            //32-bit
…
for (i=0; i<10; i++)
        A = A + 15;
```

- i @ 800
- A @ 808

Examples: For Loop

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

.text
        ADD     R1, R0, R0          # i <- 0
        ADDI    R2, R0, 10          # R2 <- 10
        LW      R3, 808(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;
```

Examples: Looping Over Arrays

- 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

Examples: Looping Over Arrays

```
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
```

Examples: Find Max

- 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

Examples: Find Max

```
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
```

Examples

- 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
```

Examples

```
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)
```

Examples: Single Pairs

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

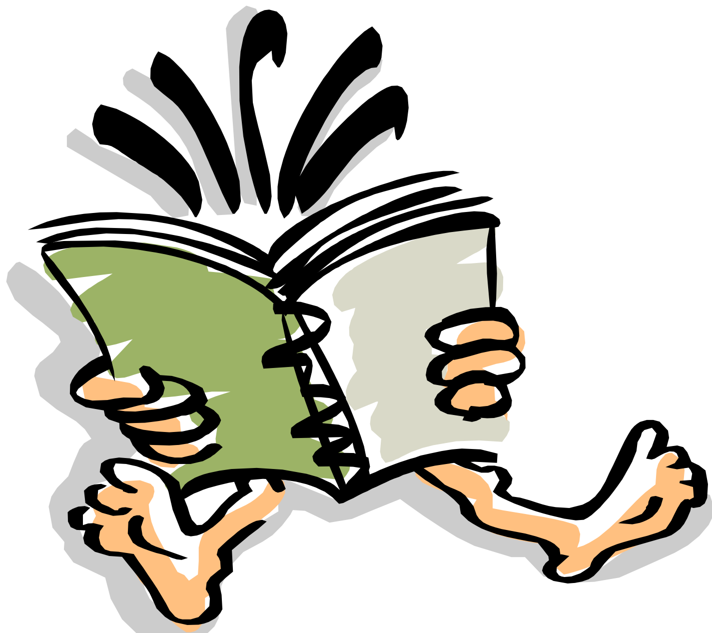
```
L.D       F1, 80(R0)          # F0 <- (A1, A2)
L.D       F2, 160(R0)         # F2 <- (B1, B2)

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

Readings



- H&P CA
  - App A
  - App K