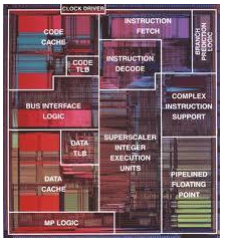
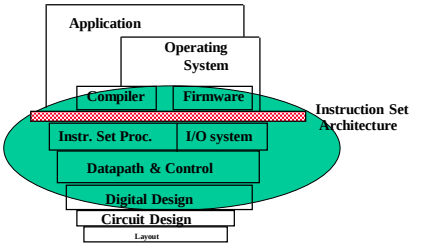
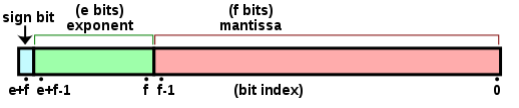
CS/SE 3340
Computer Architecture

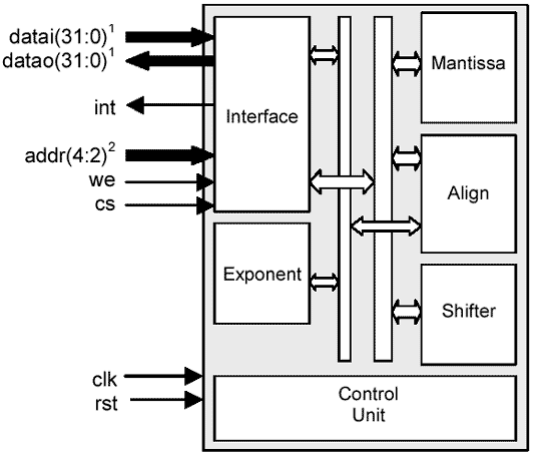




Floating Point Arithmetic

Adapted from "Computer Organization and Design, 4th Ed." by D. Patterson and J. Hennessy





http://www.altera.com/products/ip/dsp/arithmetic/m-dcd_dfpau.html

Addition

- Consider a 4-digit decimal example
  - $9.999 \times 10^{1} + 1.610 \times 10^{-1}$
- Step 1: Align decimal points
  - Shift number with smaller exponent
  - $9.999 \times 10^{1} + 0.016 \times 10^{1}$
- Step 2: Add significands
  - $9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$
- Step 3: Normalize result & check for over/underflow
  - 1.0015×10^{2}
- Step4: Round and renormalize if necessary
  - 1.002×10^{2}

3

Addition – cont'd

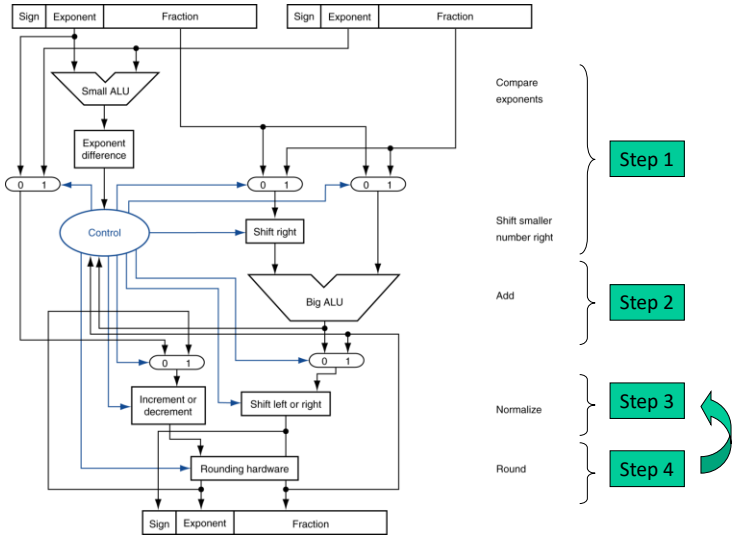
- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$
- Step 1: Align binary points
  - Shift number with smaller exponent
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- Step 2: Add significands
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
- Step 3: Normalize result & check for over/underflow
  - $1.000_2 \times 2^{-4}$, with no over/underflow
- Step 4: Round and renormalize if necessary
  - $1.000_2 \times 2^{-4}$ (no change)

4

FP Adder Hardware

- Much more complex than integer adder
- Operations take too long for one clock cycle
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be *pipelined*
  - Exploit *sub-instruction level parallelism*

FP Adder Hardware



Floating-Point Multiplication

- Consider a 4-digit decimal example
  - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- Step 1: Add exponents
  - For biased exponents, subtract bias from sum
  - New exponent = 10 + –5 = 5
- Step 2: Multiply significands
  - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^{5}$
- Step 3: Normalize result & check for over/underflow
  - 1.0212×10^{6}
- Step 4: Round and renormalize if necessary
  - 1.021×10^{6}
- Step 5: Determine sign of result from signs of operands
  - $+1.021 \times 10^{6}$

Floating-Point Multiplication

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$
- Step 1: Add exponents
  - Unbiased: –1 + –2 = –3
  - Biased: (–1 + 127) + (–2 + 127) = –3 + 254 – 127 = –3 + 127
- Step 2: Multiply significands
  - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
- Step 3: Normalize result & check for over/underflow
  - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- Step 4: Round and renormalize if necessary
  - $1.110_2 \times 2^{-3}$ (no change)
- Step 5: Determine sign
  - $-1.110_2 \times 2^{-3}$

FP Multiplier Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP ↔ integer conversion
- Operations usually takes several cycles
  - Can be pipelined

9

FP Instructions in MIPS

- FP hardware is coprocessor 1
  - Adjunct processor that extends the ISA
- Separate FP registers
  - 32 single-precision: `$f0, $f1, … $f31`
  - Paired for double-precision: `$f0/$f1, $f2/$f3,…`
    - Release 2 of MIPs ISA supports 32 × 64-bit FP reg's
- FP instructions operate only on FP registers
  - Programs generally don't do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact
- FP load and store instructions
  - `lwc1, ldc1, swc1, sdc1`
    - `e.g., ldc1 $f8, 32($sp)`

10

FP Instructions in MIPS

- Single-precision arithmetic
  - `add.s, sub.s, mul.s, div.s`
  - `e.g., add.s $f0, $f1, $f6`
- Double-precision arithmetic
  - `add.d, sub.d, mul.d, div.d`
  - `e.g., mul.d $f4, $f4, $f6`
- Single- and double-precision comparison
  - `c.xx.s, c.xx.d (xx is eq, lt, le, …)`
  - Sets or clears FP condition-code bit
    - `e.g. c.lt.s $f3, $f4`
- Branch on FP condition code true or false
  - `bc1t, bc1f`
  - `e.g., bc1t TargetLabel`

11

FP Example: °F to °C

- C code:

```
float f2c (float fahr) {
  return ((5.0/9.0)*(fahr - 32.0));
}
```

  - fahr in $f12, result in $f0, literals in global memory space
- Compiled MIPS code:

```
f2c: lwc1  $f16, const5($gp)
     lwc1  $f18, const9($gp)
     div.s $f16, $f16, $f18
     lwc1  $f18, const32($gp)
     sub.s $f18, $f12, $f18
     mul.s $f0,  $f16, $f18
     jr    $ra
```

12

FP Example: Array Multiplication

- X = X + Y × Z
  - All 32 × 32 matrices, 64-bit double-precision elements
- C code:

```
void mm (double x[][],
         double y[][], double z[][]) {
  int i, j, k;
  for (i = 0; i! = 32; i = i + 1)
    for (j = 0; j! = 32; j = j + 1)
      for (k = 0; k! = 32; k = k + 1)
        x[i][j] = x[i][j]
                  + y[i][k] * z[k][j];
}
```

  - Addresses of x, y, z in $a0, $a1, $a2, and i, j, k in $s0, $s1, $s2

FP Example: Array Multiplication

- MIPS code:

```
    li   $t1, 32       # $t1 = 32 (row size/loop end)
    li   $s0, 0        # i = 0; initialize 1st for loop
L1: li   $s1, 0        # j = 0; restart 2nd for loop
L2: li   $s2, 0        # k = 0; restart 3rd for loop
    sll  $t2, $s0, 5   # $t2 = i * 32 (size of row of x)
    addu $t2, $t2, $s1 # $t2 = i * size(row) + j
    sll  $t2, $t2, 3   # $t2 = byte offset of [i][j]
    addu $t2, $a0, $t2 # $t2 = byte address of x[i][j]
    l.d  $f4, 0($t2)   # $f4 = 8 bytes of x[i][j]
L3: sll  $t0, $s2, 5   # $t0 = k * 32 (size of row of z)
    addu $t0, $t0, $s1 # $t0 = k * size(row) + j
    sll  $t0, $t0, 3   # $t0 = byte offset of [k][j]
    addu $t0, $a2, $t0 # $t0 = byte address of z[k][j]
    l.d  $f16, 0($t0)  # $f16 = 8 bytes of z[k][j]
    …
```

FP Example: Array Multiplication

```
...
    sll  $t0, $s0, 5       # $t0 = i*32 (size of row of y)
    addu  $t0, $t0, $s2    # $t0 = i*size(row) + k
    sll   $t0, $t0, 3      # $t0 = byte offset of [i][k]
    addu  $t0, $a1, $t0    # $t0 = byte address of y[i][k]
    l.d   $f18, 0($t0)     # $f18 = 8 bytes of y[i][k]
    mul.d $f16, $f18, $f16 # $f16 = y[i][k] * z[k][j]
    add.d $f4, $f4, $f16   # f4=x[i][j] + y[i][k]*z[k][j]
    addiu $s2, $s2, 1      # $k k + 1
    bne   $s2, $t1, L3     # if (k != 32) go to L3
    s.d   $f4, 0($t2)      # x[i][j] = $f4
    addiu $s1, $s1, 1      # $j = j + 1
    bne   $s1, $t1, L2     # if (j != 32) go to L2
    addiu $s0, $s0, 1      # $i = i + 1
    bne   $s0, $t1, L1     # if (i != 32) go to L1
```


x86 FP Architecture

- Originally based on 8087 FP coprocessor
  - 8 × 80-bit extended-precision registers
  - Used as a push-down stack
  - Registers indexed from TOS: ST(0), ST(1), …
- FP values are 32-bit or 64 in memory
  - Converted on load/store of memory operand
  - Integer operands can also be converted on load/store
- Very difficult to generate and optimize code
  - Result: poor FP performance

x86 FP Instructions

Data transfer	Arithmetic	Compare	Transcendental
FILD mem/ST(i) FISTP mem/ST(i) FLDPI FLD1 FLDZ	FIADDP mem/ST(i) FISUBRP mem/ST(i) FIMULP mem/ST(i) FIDIVRP mem/ST(i) FSQRT FABS FRNDINT	FICOMP FIUCOMP FSTSW AX/mem	FPATAN F2XMI FCOS FPTAN FPREM FPSIN FYL2X

- Optional variations
  - I: integer operand
  - P: pop operand from stack
  - R: reverse operand order
  - But not all combinations allowed

17

Streaming SIMD Extension 2 (SSE2)

- Adds 4 × 128-bit registers
  - Extended to 8 registers in AMD64/EM64T
- Can be used for multiple FP operands
  - 2 × 64-bit double precision
  - 4 × 32-bit single precision
  - Instructions operate on them simultaneously
    - Single-Instruction Multiple-Data

18

Accurate Arithmetic

- IEEE Std 754 specifies additional rounding control
  - Extra bits of precision (guard, round, sticky)
  - Choice of rounding modes
  - Allows programmer to fine-tune numerical behavior of a computation
- Not all FP units implement all options
  - Most programming languages and FP libraries just use defaults
- Trade-off between hardware complexity, performance, and market requirements

19

Who Cares About FP Accuracy?

- Important for scientific code
  - But for everyday consumer use?
    - "My bank balance is out by 0.0002¢!" ☹
- The Intel Pentium FDIV bug
  - The market expects accuracy
  - See Colwell, *The Pentium Chronicles*

20

Interpretation of Data

- Bits have no inherent meaning
  - Interpretation depends on the instructions applied
- Computer representations of numbers
  - Finite range and precision
  - Need to account for this in programs

21

MIPS Design Principles

- Simplicity favors regularity
  - fixed size instructions
  - small number of instruction formats
  - opcode always the first 6 bits
- Smaller is faster
  - limited instruction set
  - limited number of registers in register file
  - limited number of addressing modes
- Make the common case fast
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands
- Good design demands good compromises
  - three instruction formats

22