

Administrative Matters

- Homework #3
 - due Monday, 10/22
- Midterm #1
 - Wednesday 10/24
 - Cumulative



Chapter 3

Arithmetic for Computers (continue)

Multiplication

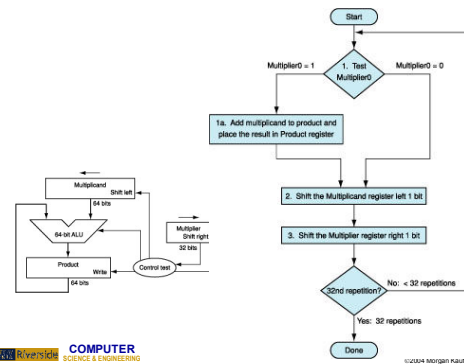
- More complicated than addition
 - accomplished via shifting and addition
- More time and more area
- Let's look at 3 versions

```

      0010 (multiplicand)
    x 1011 (multiplier)
    -----
      0010
     0000
    0000
   0010
  -----
 0010110
  
```

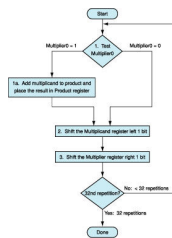
- Negative numbers: convert and multiply
 - there are better techniques, we won't look at them
 - Therefore, we will always be dealing with multiplying two positive numbers

Simple Implementation



An example

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0001	0000 0010	0000 0000
1	1a. 1 → Prod = Prod + Mxand	0011	0000 0010	0000 0010
	2. Shift left Multiplicand	0011	0000 0100	0000 0010
	3. Shift right Multiplier	0001	0000 0100	0000 0010
2	1a. 1 → Prod = Prod + Mxand	0001	0000 0100	0000 0110
	2. Shift left Multiplicand	0001	0000 1000	0000 0110
	3. Shift right Multiplier	0000	0000 1000	0000 0110
3	1. 0 → no operation	0000	0000 1000	0000 0110
	2. Shift left Multiplicand	0000	0001 0000	0000 0110
	3. Shift right Multiplier	0000	0001 0000	0000 0110
4	1. 0 → no operation	0000	0001 0000	0000 0110
	2. Shift left Multiplicand	0000	0010 0000	0000 0110
	3. Shift right Multiplier	0000	0010 0000	0000 0110

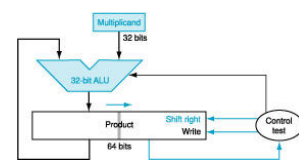


Second Implementation

- Only 32 bits ALU and Multiplicand are needed
- Multiplier 0011
- Multiplicand 00000010
- Product/Multiplier

```

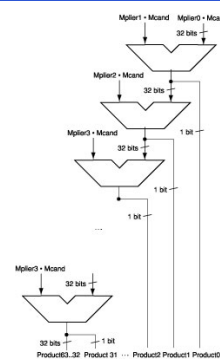
0000000000000011
0000001000000011 add
0000000100000001 shift
0000000110000000 add
0000000110000000 shift
0000000011000000 shift
0000000011000000 shift
0000000001100000 shift
0000000000011000 shift
0000000000001100 shift
0000000000000110 shift
  
```



Fastest Implementation

- “Loop Unrolling”
 - Don't have to wait for clock
 - Can be optimized further
 - Parallel multiplier

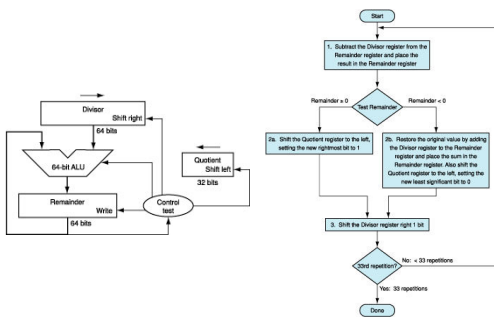
00000010
00000100
00000110



Division

Divisor 1000_{ten} 1001_{ten} Quotient
 1001010_{ten} Dividend
 -1000
 10
 101
 1010
 -1000
 10_{ten} Remainder

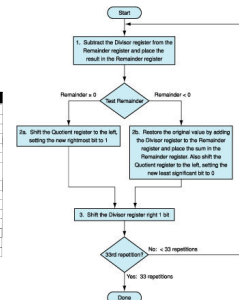
An implementation of Division



An example

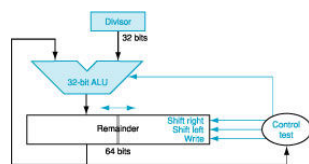
- Divide 0111 by 0010

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem - Div	0000	0010 0000	0110 0111
	2a: Rem < 0 ⇒ +Div, set Q ₀ = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem - Div	0000	0001 0000	0111 0111
	2b: Rem < 0 ⇒ +Div, set Q ₁ = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem - Div	0000	0000 1000	0000 1111
	2b: Rem < 0 ⇒ +Div, set Q ₂ = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem - Div	0001	0000 0100	0000 0011
	2b: Rem < 0 ⇒ +Div, set Q ₃ = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem - Div	0001	0000 0010	0000 0001
	2b: Rem < 0 ⇒ +Div, set Q ₄ = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001



A better Implementation

- Only 32 bit ALU and Divisor are needed
- Divide 0111 by 0010
- Quotient&Remainder
- 0000011100000000
- 1110011100000000 sub
- 0000011100000000 restore
- 0000111000000000 shift
- 1110111000000000 sub
- 0000111000000000 restore
- 0001100000000000 shift
- 1111100000000000 sub
- 0001100000000000 restore
- 0011100000000000 shift
- 0001100000000000 sub
- 0011000000000001 shift
- 0001000000000001 sub
- 0010000000000011 shift
- 0010000000000011



Floating Point Operands in MIPS

- Separate floating point register
 - Operations on integer and floating point usually don't mix
 - Double up for double precision

Name	Example	Comments
32 floating point registers	\$f0, \$f1, \$f2, ..., \$f31	MIPS floating-point registers are used in pairs for double-precision numbers.
2 ³² memory words	Memory(0), Memory(4), Memory(4294967292)	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

Floating Point Instructions in MIPS

- FP Compare sets condition
- bc1t and bc1f branch on condition
- lwc1, swc1 load/store from co-processor

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	add.s \$f2, \$f4, \$f6	$\$f2 = \$f4 + \$f6$	FP add (single precision)
	FP subtract single	sub.s \$f2, \$f4, \$f6	$\$f2 = \$f4 - \$f6$	FP sub (single precision)
	FP multiply single	mul.s \$f2, \$f4, \$f6	$\$f2 = \$f4 \times \$f6$	FP mult (single precision)
	FP divide single	div.s \$f2, \$f4, \$f6	$\$f2 = \$f4 / \$f6$	FP div (single precision)
	FP add double	add.d \$f2, \$f4, \$f6	$\$f2 = \$f4 + \$f6$	FP add (double precision)
	FP subtract double	sub.d \$f2, \$f4, \$f6	$\$f2 = \$f4 - \$f6$	FP sub (double precision)
	FP multiply double	mul.d \$f2, \$f4, \$f6	$\$f2 = \$f4 \times \$f6$	FP multiply (double precision)
	FP divide double	div.d \$f2, \$f4, \$f6	$\$f2 = \$f4 / \$f6$	FP divide (double precision)
Data transfer	load word co-processor	lwc1 \$f1, 100(\$f2)	$\$f1 = \text{Memory}[\$f2 + 100]$	32-bit data to FP register
	store word co-processor	swc1 \$f1, 100(\$f2)	$\text{Memory}[\$f2 + 100] = \$f1$	32-bit data to memory
Conditional branch	branch on FP true	bc1t \$f2, 25	if (cond == 1) go to PC + 4 + 100	PC relative branch if FP cond.
	branch on FP false	bc1f \$f2, 25	if (cond == 0) go to PC + 4 + 100	PC relative branch if not cond.
	FP compare single (eq.na, lt, gt, ge)	c.lt.s \$f2, \$f4	if ($\$f2 < \$f4$)	FP compare less than single precision
	FP compare double (eq.na, lt, gt, ge)	c.lt.d \$f2, \$f4	if ($\$f2 < \$f4$)	FP compare less than double precision

Floating Point Example

- $z = x + y$
- lwc1 \$f4, x(\$sp)
- lwc1 \$f6, y(\$sp)
- add.s \$f2, \$f4, \$f6
- swc1 \$f2, z(\$sp)

Floating Point Example

- Converting Fahrenheit to Celsius

```

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

- Load constant 5.0, 9.0, and 32.0 through $gp from heap
- Assume fahr in $f12

lwc1 $f16, const5($gp) # $f16 = 5.0 (5.0 in memory)
lwc1 $f18, const9($gp) # $f18 = 9.0 (9.0 in memory)

div.s $f16, $f16, $f18 # $f16 = 5.0 / 9.0

lwc1 $f18, const32($gp) # $f18 = 32.0
sub.s $f18, $f12, $f18 # $f18 = fahr - 32.0

mul.s $f0, $f16, $f18 # $f0 = (5/9)*(fahr - 32.0)
jr $ra # return

```

Guarded digit

- Use 2 extra bits, one guard one round
- Without it:
 - $2.56 \times 10^0 + 2.34 \times 10^2$
 - $0.02 \times 10^2 + 2.34 \times 10^2$
 - 2.36×10^2
- Using Guard and round
 - $2.56 \times 10^0 + 2.34 \times 10^2$
 - $= (0.0256 + 2.3400) \times 10^2$
 - $= 2.3656 \times 10^2$
 - which will round to 2.37×10^2
- 0-49 rounds down, 51-99 rounds up
 - 50 down 49+1 up
- Sticky bit needed to break tie at 50

A sticky situation for IEEE754

- Rounding is tough for the halfway case
- 2.2449 → 2.24
- 2.2451 → 2.25
- 2.2450? IRS always rounds up
 - Not quite fair, since there are 2.2400 to 2.2449 down 2.2451 to 2.2500 up
- 2.2450001 should be round to 2.25
- But 2.245000000 by convention rounds to 2.24
- Sticky bit to break tie
 - indication there is something at the end of 0's
- 2.245 with sticky set to 1 will round to 2.25
- 2.245 with sticky set to 0 will round to 2.24

Sticky in action

- $5.01 \times 10^{-1} + 2.34 \times 10^2$
- $(0.0050 + 2.3400) \times 10^2$
- 2.3450×10^2
 - With sticky bit set
 - Round to 2.35
- $5.00 \times 10^{-1} + 2.34 \times 10^2$
- $(0.0050 + 2.3400) \times 10^2$
- 2.3450×10^2
 - With sticky bit not set
 - Round to 2.34

Quick question

- IEEE single precision standard give
- 1 sign bit
- 8 exponent with 127 bias
- 23 significand with implied 1
- What are the possible number that I can represent?

IA-32 floating point

- Basically a stack architecture!
 - load operands from memory onto stack
 - Operation pop operands than push result onto stack
 - About 60 floating instruction
 - Plus different address mode
- 80 bits floating point on the stack
 - Still 32 bit single and 64 bit double in memory
- SSE2
 - Streaming SIMD Extension 2
 - Another 144 instructions
 - No longer stack architecture, much more like MIPS

The Floating Point Instructions & Operands of the IA-32

Data transfer	Arithmetic	Compare	Transcendental
FILD mem/ST(i)	FADD(P) mem/ST(i)	FICOMP	FPATAN
FIST(P)	FISUB(R)(P) mem/ST(i)	FUCOMP(P)(P)	F2XM1
mem/ST(i)			
FLDPI	FIMUL(P) mem/ST(i)	FSTSW AX/mem	FCOS
FLDL	FIDIV(R)(P) mem/ST(i)		FPTAN
FLDZ	FSDRT		FPREM
	FABS		FSIN
	FRRNDINT		FYL2X

Instruction	Operands	Comment
FADD		Both operands in stack; result replaces top of stack.
FADD	ST(i)	One source operand is <i>i</i> th register below the top of stack; result replaces the top of stack.
FADD	ST(i), ST	One source operand is the top of stack; result replaces <i>i</i> th register below the top of stack.
FADD	mem32	One source operand is a 32-bit location in memory; result replaces the top of stack.
FADD	mem64	One source operand is a 64-bit location in memory; result replaces the top of stack.