CSE471—Computer Design and Organization

Prof. Steven Burns *Autumn 1994*

Lecture Notes: Eighth Day

Multiplication

- Grade school multiplication

$$
\begin{array}{ccccccccccc}
 & & & & & 0 & 1 & 0 & 1 & 0 & 1 \\
 & & & & & 0 & 1 & 0 & 1 & 1 & 0 \\
\hline
 & & & & & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & 0 & 1 & 0 & 1 & 0 & 1 & \\
 & & & 0 & 1 & 0 & 1 & 0 & 1 & & \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & & & \\
 & 0 & 1 & 0 & 1 & 0 & 1 & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & & & & & \\
\hline
0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0
\end{array}
$$

- We'll develop an algorithm for Serial-Parallel Unsigned Multiplication.

Simple Algorithm for Multiplication

- The *multiplicand* and *multiplier* are both 32 bits. The result is 64-bits.
- We'll load the *multiplicand* into a 64-bit register and use two shifters and a 64-bit adder.

```
reg [63:0] a,s;
reg [31:0] b;
a = multiplicand;
b = multiplier;
s = 64'b0;
repeat (32)
begin
   if ( b[0] == 1'b1)
      s = s + a;
   a = a<<1;
   b = b>>1;
end
```

Second Algorithm for Multiplication

- We'll add the *multiplicand* into the high-order portion of the 64-bit result register. We can then use two shifters and a 32-bit adder.

```
reg [63:0] s;
reg [31:0] a,b;
a = multiplicand;
b = multiplier;
s = 64'b0;
repeat (32)
begin
   if ( b[0] == 1'b1)
      s[63:32] = s[63:32] + a;
   s = s>>1;
   b = b>>1;
end
```

Third Algorithm for Multiplication

- We'll load the *multiplicand* into a high-order portion of the 64-bit result register. We'll load the multiplier into the low-order portion of the result register. We use one shifter (64-bits) and a 32-bit adder.

```
reg [63:0] s;
reg [31:0] a;
a = multiplicand;
s = { 32'b0, multiplier};
repeat (32)
begin
   if ( s[0] == 1'b1)
      s[63:32] = s[63:32] + a;
   s = s>>1;
end
```

Signed Multiplication

- If you are willing to convert to a sign and magnitude representation, then you can just use the previous algorithm.

- If you want to stick to twos-complement, you can use Booth recoding.

- Booth invented this trick to perform more shifting and less additions during multiplications.
- The key relation is:

$$2^i + 2^{i-1} + \ldots + 2^j \quad = \quad 2^{i+1} - 2^j$$

- Each series of ones (called a run) is replaced by one addition and one subtraction.

Booth's Algorithm

- We can decide when to add or subtract by looking at the multiplier two bits at a time.
- There are four cases:
  1. `00`: No operation (middle of a run of 0s.)
  2. `01`: Add the multiplicand (end of a string of 1s.)
  3. `10`: Subtract the multiplicand (beginning of a string of 1s.)
  4. `11`: No operation (middle of a run of 1s.)

Verilog for Booth's Algorithm

- Verilog Code:

```
reg [63:0] s;
reg [31:0] a;
reg tail;
a = multiplicand;
s = { 32'b0, multiplier};
tail = 1'b0;
repeat (32)
begin
   if      ( {s[0],tail} == 2'b01 )
      s[63:32] = s[63:32] + a;
   else if ( {s[0],tail} == 2'b10 )
      s[63:32] = s[63:32] - a;
   {s,tail} = {s[63],s};
end
```

Justification For Twos-Complement Case

- This is what Booth's algorithm does:

$$\begin{array}{rl} & -a_0 \times b \\ + & (a_0 - a_1) \times b \times 2^1 \\ + & (a_1 - a_2) \times b \times 2^2 \\ + & (a_2 - a_3) \times b \times 2^3 \\ & \vdots \\ + & (a_{30} - a_{31}) \times b \times 2^{31} \end{array}$$

- Since $-a_i \times 2^i + a_i \times 2^{i+1} = a_i \times 2^i$, we can perform the sum above, factor out b and get

$$\begin{array}{l} b \times (a_{31} \times (-2^{31}) + a_{30} \times 2^{30} + \ldots + \\ a_1 \times 2^1 + a_0 \times 2^0) \end{array}$$

  which is exactly what we want.

First Cut at Division

- Only unsigned division. Just the opposite of multiplication.

```
reg [63:0] r, d;
reg [31:0] q;
r = {32'b0,dividend};
d = {divisor,32'b0};
q = 32'b0;
repeat (33)
begin
   r = r - d;
   if ( r[63] == 1'b1)
   begin
      r = r + d;
      q = {q[30:0],1'b0};
   end else
      q = {q[30:0],1'b1};
   d = d>>1;
end
remainder = r[31:0];
```

Second Cut at Division

```
reg [63:0] r;
reg [31:0] q, d;
r = {32'b0,dividend};
d = divisor;
q = 32'b0;
repeat (32)
begin
   r = r<<1;
   r[63:32] = r[63:32] - d;
   if ( r[63] == 1'b1)
   begin
      r[63:32] = r[63:32] + d;
      q = {q[30:0],1'b0};
   end else
      q = {q[30:0],1'b1};
end
remainder = r[63:32];
```

Final Cut at Division

```
reg [63:0] r;
reg [31:0] d;
r = {32'b0,dividend};
d = divisor;
repeat (32)
begin
   r = r<<1;
   r[63:32] = r[63:32] - d;
   if ( r[63] == 1'b1)
   begin
      r[63:32] = r[63:32] + d;
      r[0] = 1'b0;
   end else
      r[0] = 1'b1;
end
remainder = r[63:32];
quotient = r[31:0];
```

Finishing Touches

- Signed division is done by converting to sign and magnitude form.

- Remember that the sign of the remainder should be the same as the sign of the quotient.
- Hardware support for multiply and divide is essentially the same.
- MIPS processors provide two special registers named `hi` and `lo`. There are instructions to move values to and from these registers: `mtlo`, `mthi`, `mflo`, `mfhi`.