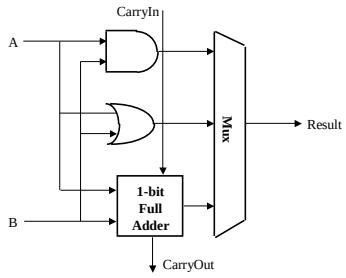
ECE468
Computer Organization & Architecture

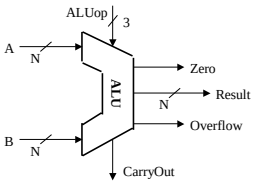
ALU Design II

Review: A One Bit ALU

- This 1-bit ALU will perform AND, OR, and ADD



Review: Functional Specification of the ALU



- ALU Control Lines (ALUop) — Function
  - 000 — And
  - 001 — Or
  - 010 — Add
  - 110 — Subtract
  - 111 — Set-on-less-than

Deriving requirements of ALU

- Start with instruction set architecture: must be able to do all operations in ISA
- Tradeoffs of cost and speed based on frequency of occurrence, hardware budget
- MIPS ISA

MIPS arithmetic instructions

Instruction	*Example*	*Meaning*	*Comments*
add	add $1,$2,$3	$1 = $2 + $3	3 operands; exception possible
subtract	sub $1,$2,$3	$1 = $2 – $3	3 operands; exception possible
add immediate	addi $1,$2,100	$1 = $2 + 100	+ constant; exception possible
add unsigned	addu $1,$2,$3	$1 = $2 + $3	3 operands; no exceptions
subtract unsigned	subu $1,$2,$3	$1 = $2 – $3	3 operands; no exceptions
add imm. unsign.	addiu $1,$2,100	$1 = $2 + 100	+ constant; no exceptions
multiply	mult $2,$3	Hi, Lo = $2 x $3	64-bit signed product
multiply unsigned	multu$2,$3	Hi, Lo = $2 x $3	64-bit unsigned product
divide	div $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Lo = quotient, Hi = remainder
divide unsigned	divu $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Unsigned quotient & remainder
Move from Hi	mfhi $1	$1 = Hi	Used to get copy of Hi
Move from Lo	mflo $1	$1 = Lo	Used to get copy of Lo

MIPS logical instructions

Instruction	*Example*	*Meaning*	*Comment*
and	and $1,$2,$3	$1 = $2 & $3	3 reg. operands; Logical AND
or	or $1,$2,$3	$1 = $2 | $3	3 reg. operands; Logical OR
xor	xor $1,$2,$3	$1 = $2 ⊕ $3	3 reg. operands; Logical XOR
nor	nor $1,$2,$3	$1 = ~($2 |$3)	3 reg. operands; Logical NOR
and immediate	andi $1,$2,10	$1 = $2 & 10	Logical AND reg, constant
or immediate	ori $1,$2,10	$1 = $2 | 10	Logical OR reg, constant
xor immediate	xori $1, $2,10	$1 = ~$2 &~10	Logical XOR reg, constant
shift left logical	sll $1,$2,10	$1 = $2 << 10	Shift left by constant
shift right logical	srl $1,$2,10	$1 = $2 >> 10	Shift right by constant
shift right arithm.	sra $1,$2,10	$1 = $2 >> 10	Shift right (sign extend)
shift left logical	sllv $1,$2,$3	$1 = $2 << $3	Shift left by variable
shift right logical	srlv $1,$2, $3	$1 = $2 >> $3	Shift right by variable
shift right arithm.	srav $1,$2, $3	$1 = $2 >> $3	Shift right arith. by variable

Compare and Branch

- Compare and Branch
  - BEQ rs, rt, offset if R[rs] == R[rt] then PC-relative branch
  - BNE rs, rt, offset <>
- Compare to zero and branch
  - BLEZ rs, offset if R[rs] <= 0 then PC-relative branch
  - BGTZ rs, offset >
  - BLT <
  - BGEZ >=
  - BLTZAL rs, offset if R[rs] < 0 then branch and link (into R 31)
  - BGEZAL >=

MIPS ALU requirements

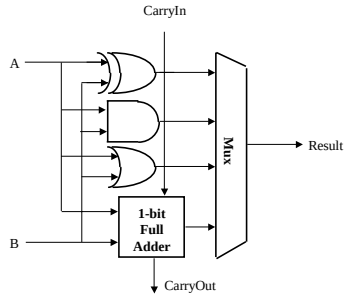
- Add, AddU, Sub, SubU, AddI, AddIU
  => 2's complement adder with overflow detection & inverter
- SLTI, SLTIU (set less than)
  => 2's complement adder with inverter, check sign bit of result
- BEQ, BNE (branch on equal or not equal)
  => 2's complement adder with inverter, check if result = 0
- And, Or, AndI, OrI
  => Logical AND, logical OR
- ALU from last lecture supports these ops

Additional MIPS ALU requirements

- Xor, Nor, XorI
  => Logical XOR, logical NOR or use 2 steps: (A OR B) XOR 1111....1111
- Sll, Srl, Sra
  => Need left shift, right shift, right shift arithmetic by 0 to 31 bits
- Mult, MultU, Div, DivU
  => Need 32-bit multiply and divide, signed and unsigned

Add XOR to ALU

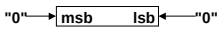
- Expand Multiplexor



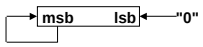
Shifters

Three different kinds:

logical-- value shifted in is always "0"



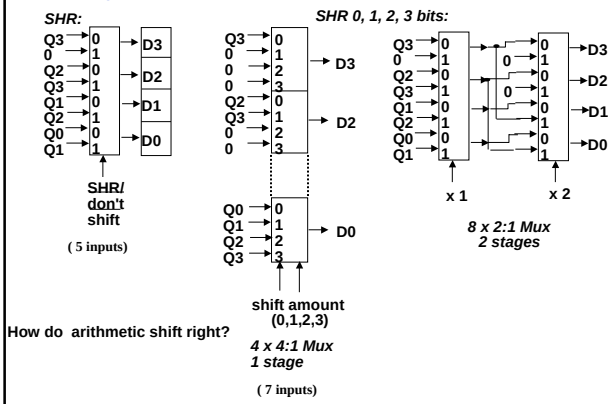
arithmetic-- on right shifts, sign extend



rotating-- shifted out bits are wrapped around (not in MIPS)

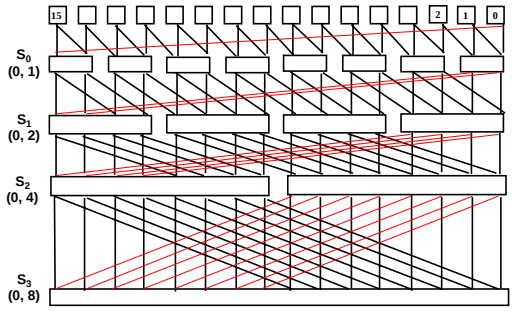
Note: these are single bit shifts. A given instruction might request 0 to 32 bits to be shifted!

Multiplexor/Shifter



How do arithmetic shift right?

General Shift Right Scheme using 16 bit example



- Shamt = $S_3S_2S_1S_0$; If S_iS=0, go straight; if S_i=0, go skew.
- If right-to-left connections are added, it could support Rotate Shift Right.

MULTIPLY (p250)

° Paper and pencil example:

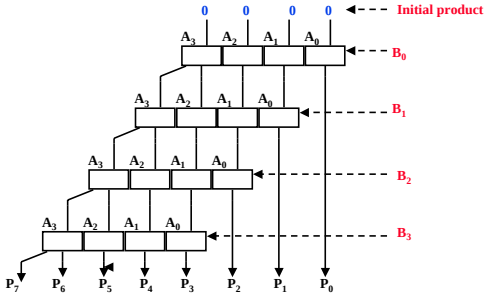
```
Multiplicand        1000
Multiplier      x   1001
                    1000
                   0000
                  0000
                 1000
Product          1001000
```

° m bits x n bits = m+n bit product

° Binary makes it easy: only 2 choices at each step
- 1 => place multiplicand (1 x multiplicand)
- 0 => place 0 (0 x multiplicand)

° 3 versions of multiply hardware & algorithm: successive refinement

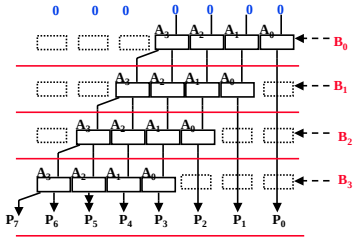
Unsigned Combinational Multiplier



° Stage i accumulates $A * 2^i$ if $B_i == 1$

° Q: How much hardware for 32 bit multiplier? Critical path?

How does it work?



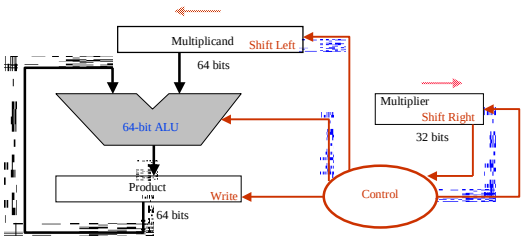
° at each stage shift A left (x 2)

° use next bit of B to determine whether to add in shifted multiplicand

° accumulate 2n bit partial product at each stage

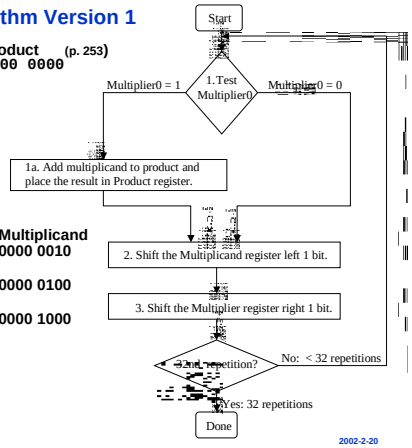
MULTIPLY HARDWARE Version 1

° 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit multiplier reg



Multiply Algorithm Version 1

Multiplier Multiplicand Product (p. 253)
0011 0000 0010 0000 0000

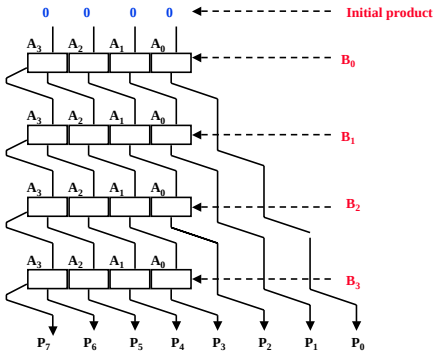


	Product	Multiplier	Multiplicand
°	0000 0000	0011	0000 0010
°	0000 0010	0001	0000 0100
°	0000 0110	0000	0000 1000
°	0000 0110		

Observations on Multiply Version 1

- 1 clock cycle per step => ~ 100 cycles per multiply of two 32-bits.
  - Ratio of multiply to add 1:5 to 1:100.
  - Amdahl's Law: even a moderate frequency of a slow operation can limit performance.
- 1/2 bits in multiplicand always 0
  => 64-bit adder is wasted
- 0 is inserted in left of multiplicand as shifted
  => least significant bits of product never changed once formed
- Very big, too slow.
- Instead of shifting multiplicand to left, shift product to right?

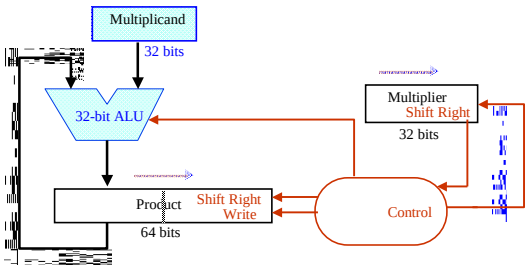
What's going on?



- Multiplicand stays still and product moves right

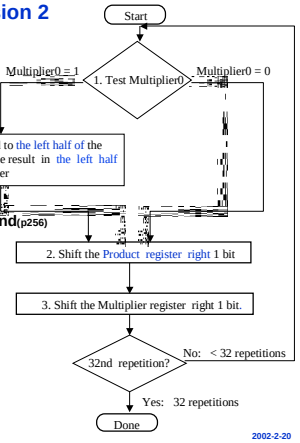
MULTIPLY HARDWARE Version 2

- 32-bit Multiplicand reg, 32 -bit ALU, 64-bit Product reg, 32-bit Multiplier reg



Multiply Algorithm Version 2

- Addition performed only on left half of product register
- Shift of product Register



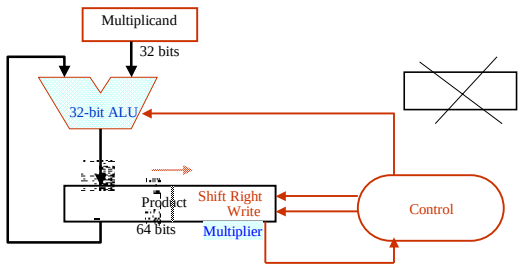
Product	Multiplier	Multiplicand(p256)
0000 0000	0011	0010
0010 0000	0011	0010
0001 0000	0001	0010
0011 0000	0001	0010
0001 1000	0000	0010
0000 1100	0000	0010
0000 0110	0000	0010

Observations on Multiply Version 2

- Product register wastes space that exactly matches size of multiplier
  => combine Multiplier register and Product register

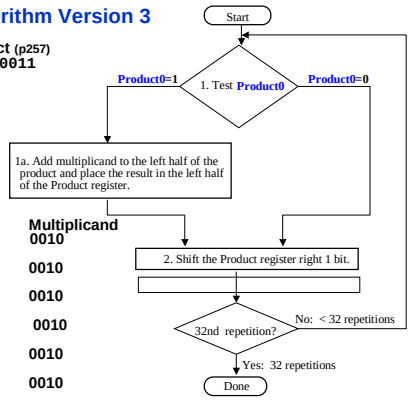
MULTIPLY HARDWARE Version 3

- 32-bit Multiplicand reg, 32 -bit ALU, 64-bit Product reg, (0-bit Multiplier reg)



Multiply Algorithm Version 3

Multiplicand 0010 Product (p257) 0000 0011



Product | Multiplier	Multiplicand
0000 0011	0010
0010 0011	0010
0001 0001	0010
0011 0000	0010
0001 1000	0010
0000 1100	0010
0000 0110	0010

Observations on Multiply Version 3

- 2 steps per bit because Multiplier & Product combined
- MIPS registers Hi and Lo are left and right half of Product
- Gives us MIPS instruction MultU
- What about signed multiplication?
  - easiest solution is to make both positive & remember whether to complement product when done (leave out the sign bit, run for 31 steps)
  - Booth's Algorithm is more elegant way to multiply signed numbers using same hardware as before

Motivation for Booth's Algorithm

- Example 2 x 6 = 0010 x 0110:

```
          0010
x         0110
+         0000      shift (0 in multiplier)
+        0010       add (1 in multiplier)
+       0100        add (1 in multiplier)
+      0000         shift (0 in multiplier)
       00001100
```

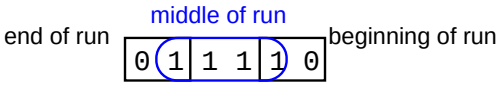
- ALU with add or subtract gets same result in more than one way:

```
6      = – 2 + 8          , or
0110   = – 0010 + 1000 = 1110 + 1000
```

- Replace a string of 1s in multiplier with an initial subtract when we first see a one and then later add for the bit after the last one. For example

```
          0010
x         0110
+         0000  shift (0 in multiplier)
–         0010  sub (first 1 in multiplier)
+       0000    shift (middle of string of 1s)
+      0010     add (prior step had last 1)
       00001100
```


Booth's Algorithm Insight



Current Bit	Bit to the Right	Explanation	Example
1	0	Beginning of a run of 1s	0001111000
1	1	Middle of a run of 1s	0001111000
0	1	End of a run of 1s	0001111000
0	0	Middle of a run of 0s	0001111000

Originally for Speed since shift faster than add for his machine

- Replace a string of 1s in multiplier with an initial subtract when we first see a one and then later add for the bit after the last one

```
      –1
+ 10000
 01111
```


Booth's Algorithm

1. Depending on the current and previous bits, do one of the following:

   00: a. Middle of a string of 0s, so no arithmetic operations.
   01: b. End of a string of 1s, so add the multiplicand to the left half of the product.
   10: c. Beginning of a string of 1s, so subtract the multiplicand from the left half of the product.
   11: d. Middle of a string of 1s, so no arithmetic operation.

2. As in the previous algorithm, shift the Product register right (arith) 1 bit.

Multiplicand 0010 Product (2 x 7) 0000 0111 0

Multiplicand 0010 Product (2 x -3) 0000 1101 0

Booths Example: 2 x 7 (p261)

mythical bit

Operation	Multiplicand	Product|Multiplier	next?
0. initial value	0010	0000 0111 0	10 -> sub
1a. P = P - m	1110	+ 1110	
		1110 0111 0	shift P (sign ext)
1b.	0010	1111 0011 1	11 -> nop, shift
2.	0010	1111 1001 1	11 -> nop, shift
3.	0010	1111 1100 1	01 -> add
4a.	0010	+ 0010	
		0001 1100 1	shift
4b.	0010	0000 1110 0	done

Booths Example: 2 x –3 (pp261-262)

mythical bit

Operation	Multiplicand	Product	next?
0. initial value	0010	0000 1101 0	10 -> sub
1a. P = P - m	1110	+ 1110	
		1110 1101 0	shift P (sign ext)
1b.	0010	1111 0110 1	01 -> add
		+ 0010	
2a.		0001 0110 1	shift P
2b.	0010	0000 1011 0	10 -> sub
		+ 1110	
3a.	0010	1110 1011 0	shift
3b.	0010	1111 0101 1	11 -> nop
4a		1111 0101 1	shift
4b.	0010	1111 1010 1	done

Summary

- Instruction Set drives the ALU design
- Shifter: success refinement from 1/bit at a time shift register to barrel shifter
- Multiply: successive refinement to see final design
  - 32-bit Adder, 64-bit shift register, 32-bit Multiplicand Register
  - Booth's algorithm to handle signed multiplies
- There are algorithms that calculate many bits of multiply per cycle
- What's Missing from MIPS is Divide & Floating Point Arithmetic: Next time the Pentium Bug