EEL-4713C
Computer Architecture
Introduction: the Logic Design Process

Outline of Today's Lecture

° An Overview of the Design Process
  • Illustration using example of ALU design

° Reading: Appendix C.5-C6

The Design Process

"To Design Is To Represent"

Design activity yields description/representation of an object

-- Distinguish concept from artifact

-- The concept is captured in one or more *representation languages*

-- This process IS design

Design Begins With Requirements

-- Functional Capabilities: what it will do

-- Performance Characteristics: Speed, Power, Area, Cost, . . .

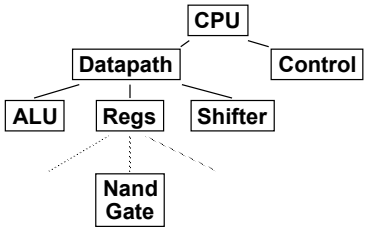
Design Process

Design Finishes As Assembly

-- Design understood in terms of components and how they have been assembled

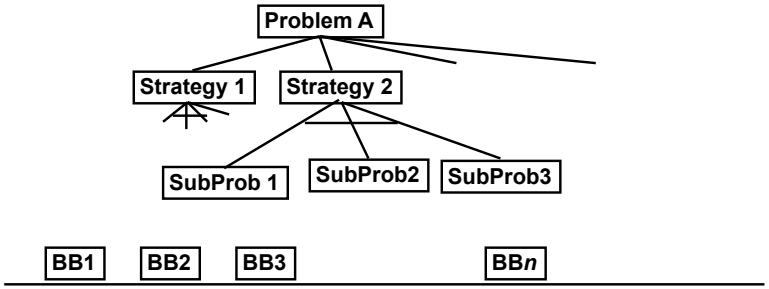
-- Top Down *decomposition* of complex functions (behaviors) into more primitive functions

-- Bottom-up *composition* of primitive building blocks into more complex assemblies



Design is a creative process, not a simple method

Design as Search



Design involves educated guesses and verification

-- Given the goals, how should these be prioritized?

-- Given alternative design pieces, which should be selected?

-- Given design space of components & assemblies, which part will yield the best solution?

Feasible (good) choices vs. Optimal choices

Problem: Design a "fast" ALU for the MIPS ISA

° Requirements?

° Must support the Arithmetic / Logic operations

° Tradeoffs of cost and speed based on frequency of occurrence, hardware budget

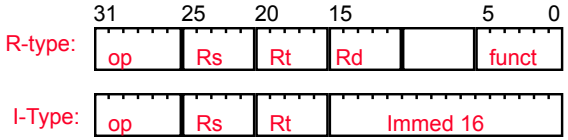
MIPS ALU requirements

° Add, AddU, Sub, SubU, Addl, AddlU
  • => 2's complement adder/sub with overflow detection

° And, Or, Andl, Orl, Xor, Xori, Nor
  • => Logical AND, logical OR, XOR, nor

° SLTI, SLTIU (set less than)
  • => 2's complement adder with inverter, check sign bit of result

MIPS arithmetic instruction format



Type	op	funct
ADDI	10	xx
ADDIU	11	xx
SLTI	12	xx
SLTIU	13	xx
ANDI	14	xx
ORI	15	xx
XORI	16	xx
LUI	17	xx

Type	op	funct
ADD	00	40
ADDU	00	41
SUB	00	42
SUBU	00	43
AND	00	44
OR	00	45
XOR	00	46
NOR	00	47

Type	op	funct
	00	50
	00	51
SLT	00	52
SLTU	00	53

° Signed arithmetic generates overflow, no carry

Design Trick: divide & conquer

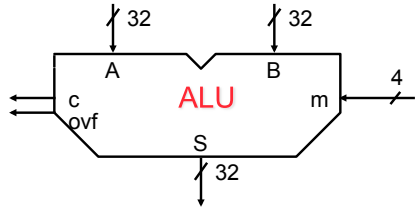
- Break the problem into simpler problems, solve them and glue together the solution
- Example: assume the immediates have been taken care of before the ALU
  - 10 operations (4 bits)

00	add
01	addU
02	sub
03	subU
04	and
05	or
06	xor
07	nor
12	slt
13	sltU

Refined Requirements

(1) Functional Specification
inputs: 2 x 32-bit operands A, B, 4-bit mode (control)
outputs: 32-bit result S, 1-bit carry, 1 bit overflow
operations: add, addu, sub, subu, and, or, xor, nor, slt, sltU

(2) Block Diagram (CAD-TOOL symbol, VHDL entity)



*Behavioral Representation: VHDL

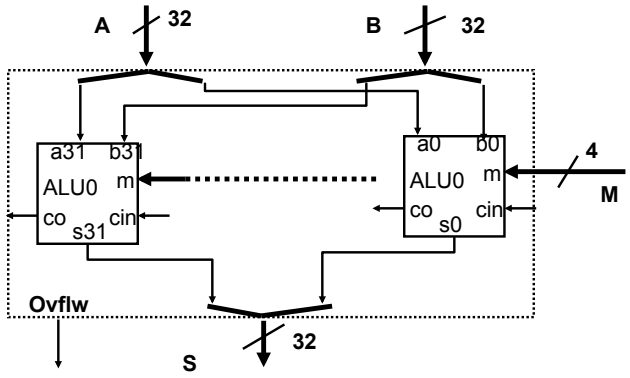
```
Entity ALU is
   generic (c_delay: integer := 20 ns;
            S_delay: integer := 20 ns);

   port (   signal A, B:  in  vlbit_vector (0 to 31);
      signal    m:  in  vlbit_vector (0 to 3);
      signal    S: out  vlbit_vector (0 to 31);
      signal    c: out  vlbit;
      signal  ovf: out  vlbit)
end ALU;

...

      S <= A + B;
```

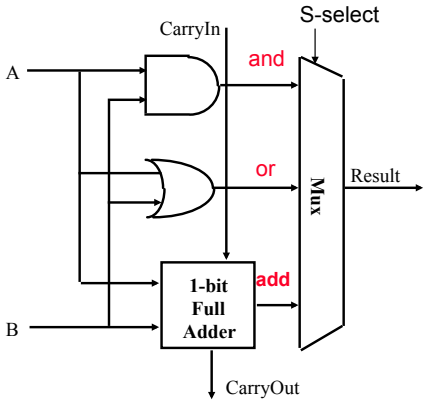

Refined Diagram: bit-slice ALU



Glue logic: selection/multiplexing

° Design trick 2: take pieces you know (or can imagine) and try to put them together

° Design trick 3: solve part of the problem and extend



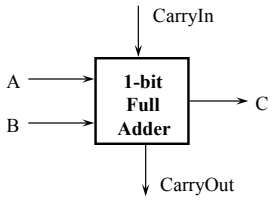
- Here is a design for a 1-bit ALU:
  - Performs AND, OR, and ADD
  - Not SUB
- Can create a 4-bit ALU by connecting 4 1-bit ALUs together
  - Carry out -> Carry in

A One-bit Full Adder

° This is also called a (3, 2) adder
  - 3 inputs, 2 outputs

° Half Adder: No CarryIn nor CarryOut

° Truth Table:



Inputs			Outputs		
A	B	CarryIn	CarryOut	Sum	Comments
0	0	0	0	0	0 + 0 + 0 = 00
0	0	1	0	1	0 + 0 + 1 = 01
0	1	0	0	1	0 + 1 + 0 = 01
0	1	1	1	0	0 + 1 + 1 = 10
1	0	0	0	1	1 + 0 + 0 = 01
1	0	1	1	0	1 + 0 + 1 = 10
1	1	0	1	0	1 + 1 + 0 = 10
1	1	1	1	1	1 + 1 + 1 = 11

Logic Equation for CarryOut

Inputs			Outputs		
A	B	CarryIn	CarryOut	Sum	Comments
0	0	0	0	0	0 + 0 + 0 = 00
0	0	1	0	1	0 + 0 + 1 = 01
0	1	0	0	1	0 + 1 + 0 = 01
→ 0	1	1	1	0	0 + 1 + 1 = 10
1	0	0	0	1	1 + 0 + 0 = 01
→ 1	0	1	1	0	1 + 0 + 1 = 10
→ 1	1	0	1	0	1 + 1 + 0 = 10
→ 1	1	1	1	1	1 + 1 + 1 = 11

° CarryOut = (!A & B & CarryIn) | (A & !B & CarryIn) | (A & B & !CarryIn) | (A & B & CarryIn)

° CarryOut = B & CarryIn | A & CarryIn | A & B

Logic Equation for Sum

Inputs			Outputs		
A	B	CarryIn	CarryOut	Sum	Comments
0	0	0	0	0	0 + 0 + 0 = 00
→ 0	0	1	0	1	0 + 0 + 1 = 01
→ 0	1	0	0	1	0 + 1 + 0 = 01
0	1	1	1	0	0 + 1 + 1 = 10
→ 1	0	0	0	1	1 + 0 + 0 = 01
1	0	1	1	0	1 + 0 + 1 = 10
1	1	0	1	0	1 + 1 + 0 = 10
→ 1	1	1	1	1	1 + 1 + 1 = 11

° Sum = (!A & !B & CarryIn) | (!A & B & !CarryIn) | (A & !B & !CarryIn) | (A & B & CarryIn)

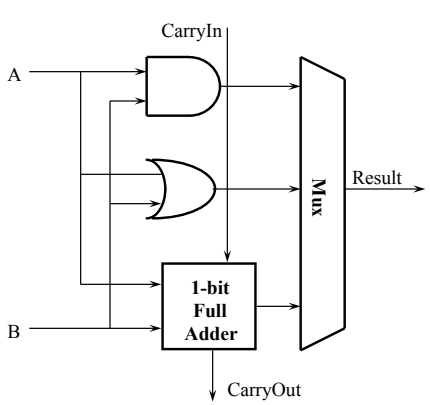
Logic Equation for Sum (continue)

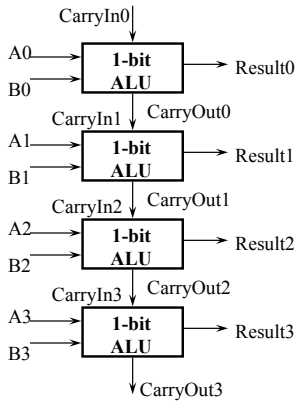
- Sum = (!A & !B & CarryIn) | (!A & B & !CarryIn) | (A & !B & !CarryIn) | (A & B & CarryIn)
- Sum = A XOR B XOR CarryIn
- Truth Table for XOR:

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

A 4-bit ALU

- 1-bit ALU 4-bit ALU

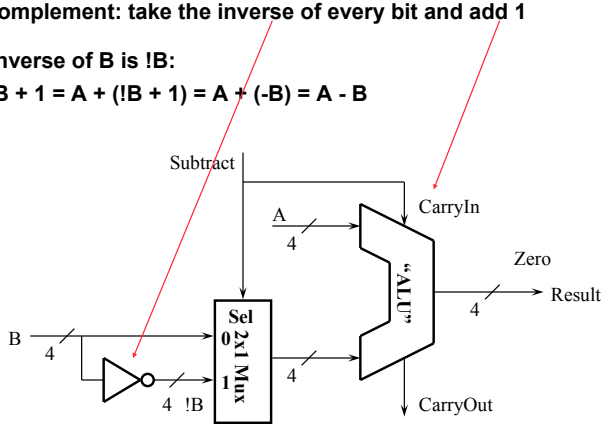




- Still no SUB!

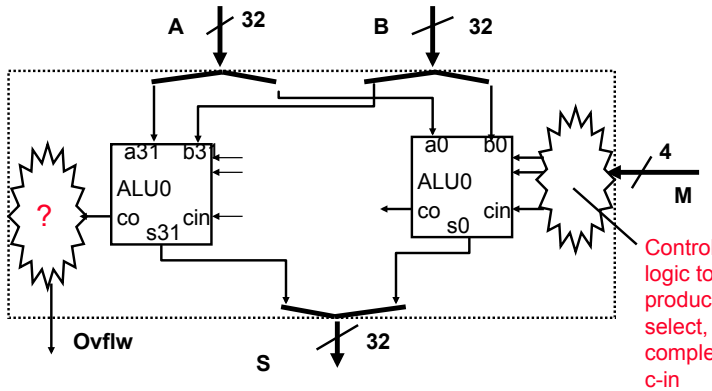
How About Subtraction?

- Keep in mind the following:
  - (A - B) is the same as: A + (-B)
  - 2's Complement: take the inverse of every bit and add 1
- Bit-wise inverse of B is !B:
  - A + !B + 1 = A + (!B + 1) = A + (-B) = A - B



Revised Diagram

- LSB and MSB need to do a little extra



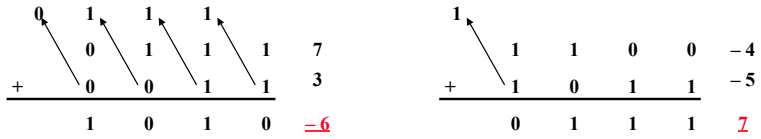
Overflow

Decimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Decimal	2's Complement
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

° Examples: 7 + 3 = 10 but ...

° - 4 - 5 = - 9 but ...



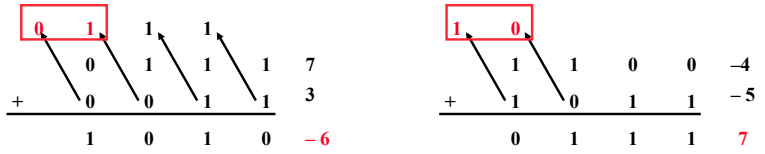
Overflow Detection

° Overflow: the result is too large (or too small) to represent properly
- 2's complement 4-bit range example: - 8 < = 4-bit binary number <= 7

° When adding operands with different signs, overflow cannot occur!

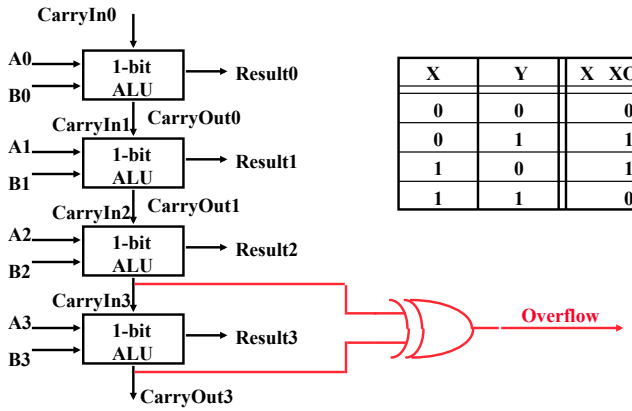
° Overflow occurs when adding:
- 2 positive numbers and the sum is negative
- 2 negative numbers and the sum is positive

° On your own: Prove you can detect overflow by:
- Carry into MSB ≠ Carry out of MSB



Overflow Detection Logic

° Carry into MSB ≠ Carry out of MSB
- For a N-bit ALU: Overflow = CarryIn[N - 1] XOR CarryOut[N - 1]

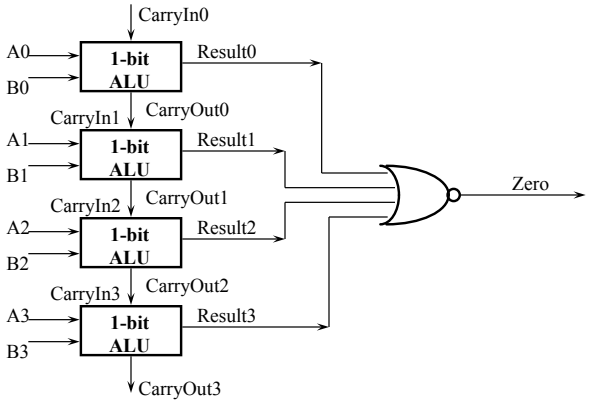


X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Zero Detection Logic

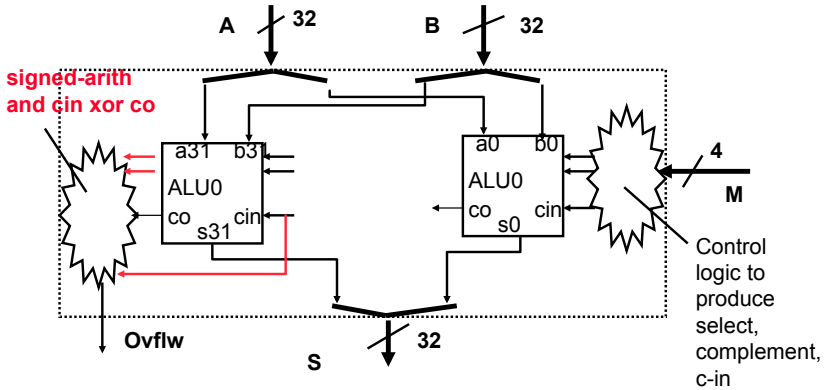
° Zero Detection Logic is just one big NOR gate
- Any non-zero input to the NOR gate will cause its output to be zero

° Leverage this for BNE (a-b != 0) and BEQ (a-b == 0)



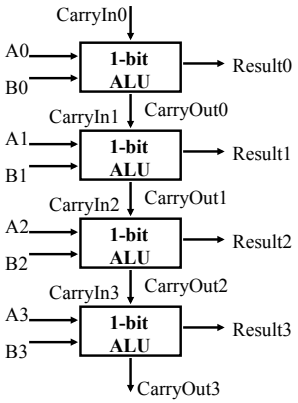
More Revised Diagram

° LSB and MSB need to do a little extra



But What about Performance?

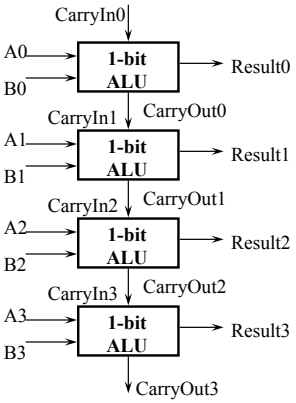
° Critical Path of n-bit Rippled-carry adder is n*CP_1bit

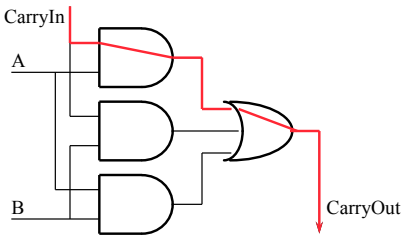


Design Trick: add hardware to deal with critical path separately

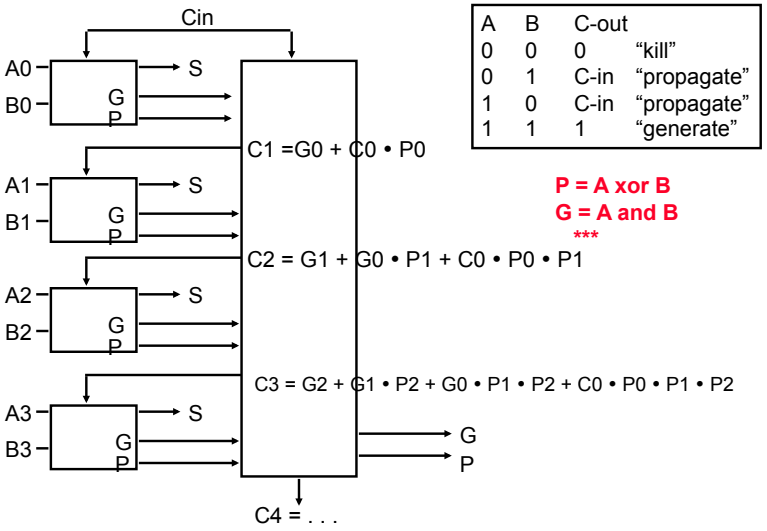
The Disadvantage of Ripple Carry

° The adder we just built is called a "Ripple Carry Adder"
- The carry bit may have to propagate from LSB to MSB
- Worst case delay for a N-bit adder: 2N-gate delay





Carry Look Ahead



A	B	C-out	
0	0	0	"kill"
0	1	C-in	"propagate"
1	0	C-in	"propagate"
1	1	1	"generate"

P = A xor B
G = A and B
***

The Idea Behind Carry Lookahead (Continue)

° Using the two new terms we just defined:
  • Generate Carry at Bit i gi = Ai & Bi
  • Propagate Carry via Bit i pi = Ai or Bi

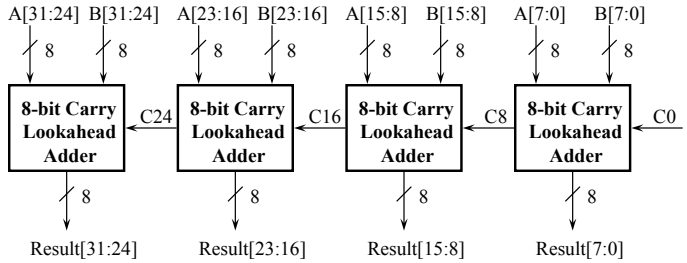
° We can rewrite:
  • Cin1 = g0 | (p0 & Cin0)
  • Cin2 = g1 | (p1 & g0) | (p1 & p0 & Cin0)
  • Cin3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & Cin0)

° Carry going into bit 3 is 1 if
  • We generate a carry at bit 2 (g2)
  • Or we generate a carry at bit 1 (g1) and bit 2 allows it to propagate (p2 & g1)
  • Or we generate a carry at bit 0 (g0) and bit 1 as well as bit 2 allows it to propagate (p2 & p1 & g0)
  • Or we have a carry input at bit 0 (Cin0) and bit 0, 1, and 2 all allow it to propagate (p2 & p1 & p0 & Cin0)

A Partial Carry Lookahead Adder

° It is very expensive to build a "full" carry lookahead adder
  • Just imagine the length of the equation for Cin31

° Common practices:
  • Connect several N-bit Lookahead Adders to form a big adder
    - Two levels of look-aheads (cascaded, as seen before)
    - Or, ripple-carry of look-aheads
  • Example: connect four 8-bit carry lookahead adders to form a 32-bit partial carry lookahead adder



Elements of the Design Process

° Divide and Conquer (e.g., ALU)
  • Formulate a solution in terms of simpler components.
  • Design each of the components (subproblems)

° Generate and Test (e.g., ALU)
  • Given a collection of building blocks, look for ways of putting them together that meets requirement

° Successive Refinement (e.g., carry lookahead)
  • Solve "most" of the problem (i.e., ignore some constraints or special cases), then examine and correct shortcomings.

Summary of the Design Process

Hierarchical Design to manage complexity

Importance of Design Representations:

Block Diagrams
Decomposition into Bit Slices
Truth Tables, K-Maps
Circuit Diagrams

top down

bottom up

mux design meets at TT

Other Descriptions: state diagrams, timing diagrams, . . .

Optimization Criteria:

Gate Count — *Area*
[Package Count] — *Pin Out*

Logic Levels — *Delay* — *Power*
Fan-in/Fan-out — *Cost* — *Design time*

Next lecture

- The MIPS single-cycle datapath
  - 4.1-4.4