L05: Instruction Set Architecture 3 & ALU Design

Adapted from:

CS152: Computer Architecture and Engineering
Dave Patterson (www.cs.berkeley.edu/~pattrsn)

Overview

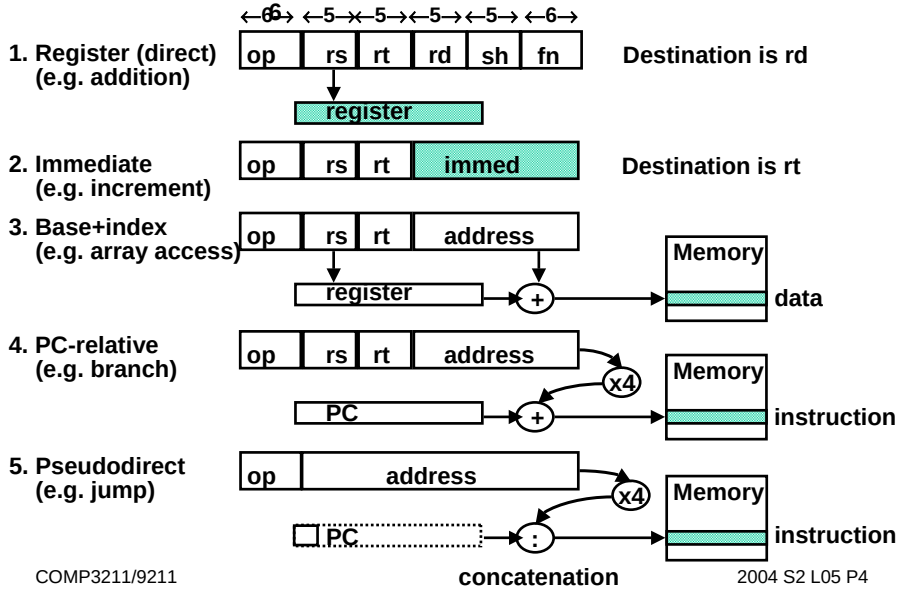
1. **Instruction Set Architecture**
- **Quick recap**
- **Conditional instructions**
- **Calling conventions**
- **Stack operations**
- **Summary**

2. **ALU Design**
- **Design Process**

Quick Recap

5 Addressing Modes/3 Instruction Formats

- **All instructions 32 bits wide**



Methods for Testing Conditions

° Condition Codes

Processor status bits are set as a side-effect of arithmetic instructions (possibly on Moves) or explicitly by compare or test instructions.

ex: add r1, r2, r3
bz label

° Compare and Branch

Ex: bgt r1, r2, label

Condition Codes

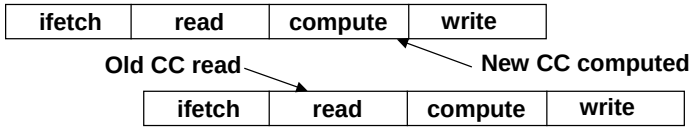
Setting CC as side effect can reduce the # of instructions

```
X:   .                          X:   .
     .                               .
     .          vs.                  .
   SUB  r0, r0, 1                  SUB  r0, r0, 1
   BRP  X                          CMP  r0, 0
                                   BRP  X
```

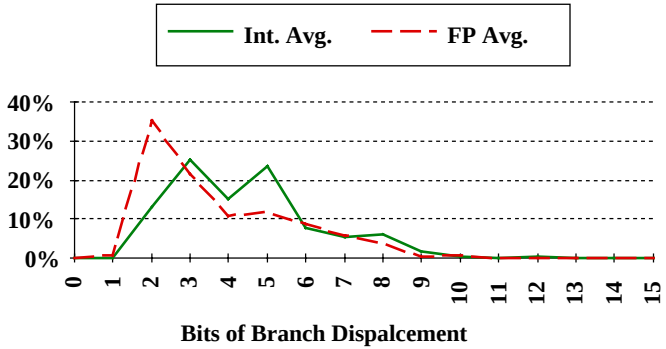
But also has disadvantages:

--- not all instructions set the condition codes; which do and which do not often confusing!
e.g., shift instruction sets the carry bit

--- dependency between the instruction that sets the CC and the one that tests it: to overlap their execution, may need to separate them with an instruction that does not change the CC



Conditional Branch Distance



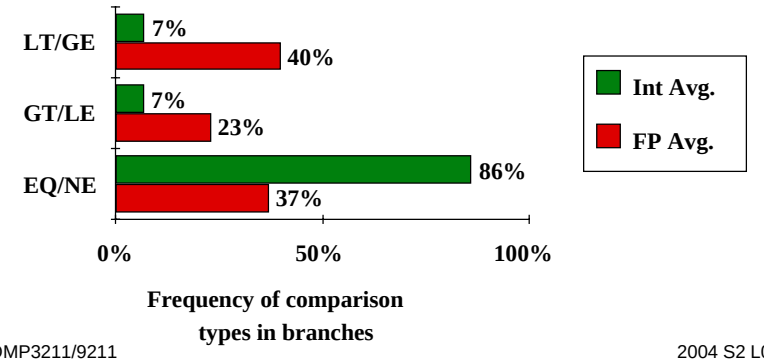
• 6 – 8 bits suffice to cover the majority of branch displacements

Conditional Branch Addressing

• PC-relative since most branches are relatively close to the current PC address

• At least 8 bits suggested (± 128 instructions)

• Compare Equal/Not Equal most important for integer programs (86%)



MIPS 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 >=**
- **Remaining set of compare and branch take two instructions**
- **Almost all comparisons are against zero!**

MIPS jump, branch, compare instructions

Instruction	*Example*	*Meaning*
branch on equal	beq $1,$2,100	if ($1 == $2) go to PC+4+400
	Equal test; PC relative branch	
branch on not eq.	bne $1,$2,100	if ($1!= $2) go to PC+4+400
	Not equal test; PC relative	
set on less than	slt $1,$2,$3	if ($2 < $3) $1=1; else $1=0
	Compare less than; 2's comp.	
set less than imm.	slti $1,$2,100	if ($2 < 100) $1=1; else $1=0
	Compare < constant; 2's comp.	
set less than uns.	sltu $1,$2,$3	if ($2 < $3) $1=1; else $1=0
	Compare less than; natural numbers	
set l. t. imm. uns.	sltiu $1,$2,100	if ($2 < 100) $1=1; else $1=0
	Compare < constant; natural numbers	
jump	j 2500	go to 10000
	Jump to target address	
jump register	jr $31	go to $31
	For switch, procedure return	
jump and link	jal 2500	$31 = PC + 4; go to 10000
	For procedure call	

Signed vs. Unsigned Comparison

Value?
2's comp Unsigned?

R1= 0…00 0000 0000 0000 0001 two
R2= 0…00 0000 0000 0000 0010 two
R3= 1…11 1111 1111 1111 1111 two

- **After executing these instructions:**

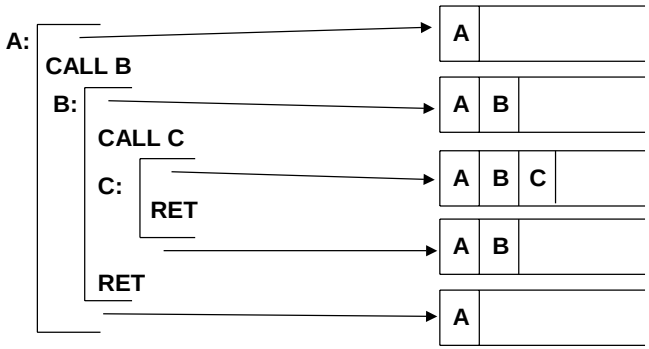
```
slt  r4,r2,r1 ;  if (r2 < r1) r4=1; else r4=0
slt  r5,r3,r1 ;  if (r3 < r1) r5=1; else r5=0
sltu r6,r2,r1 ;  if (r2 < r1) r6=1; else r6=0
sltu r7,r3,r1 ;  if (r3 < r1) r7=1; else r7=0
```

- **What are values of registers r4 - r7? Why?**

r4 = ; r5 = ; r6 = ; r7 = ;

Calls: Why Are Stacks So Great?

Stacking of Subroutine Calls & Returns and Environments:



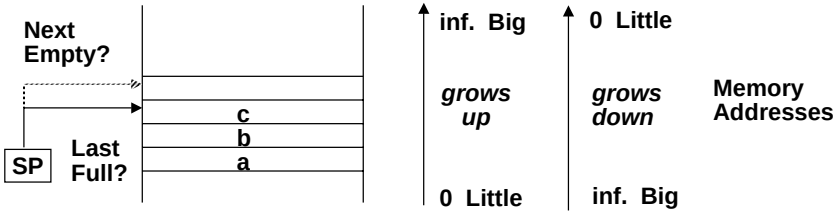
Some machines provide a memory stack as part of the architecture (e.g., VAX)

Sometimes stacks are implemented via software convention (e.g., MIPS)

Memory Stacks

Useful for stacked environments/subroutine call & return even if operand stack not part of architecture

Stacks that Grow Up vs. Stacks that Grow Down:



How is empty stack represented?

Little --> Big/Last Full

POP: Read from Mem(SP)
Decrement SP

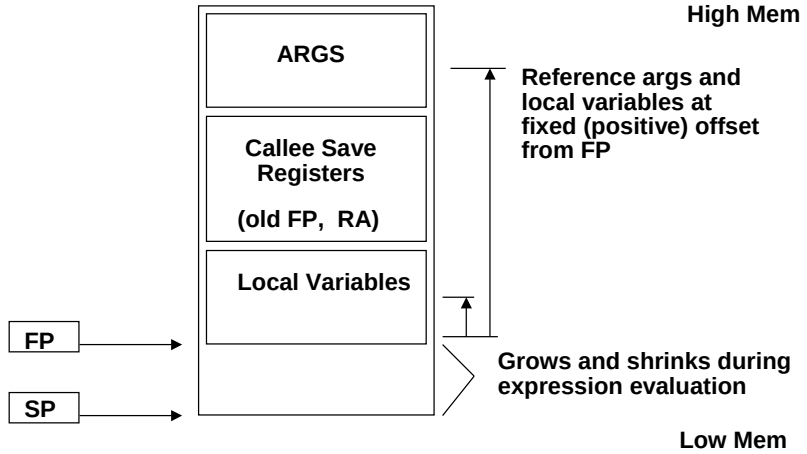
PUSH: Increment SP
Write to Mem(SP)

Little --> Big/Next Empty

POP: Decrement SP
Read from Mem(SP)

PUSH: Write to Mem(SP)
Increment SP

Call-Return Linkage: Stack Frames



- Many variations on stacks possible (up/down, last pushed / next)
- Block structured languages contain link to lexically enclosing frame
- Compilers normally keep scalar variables in registers, not memory!

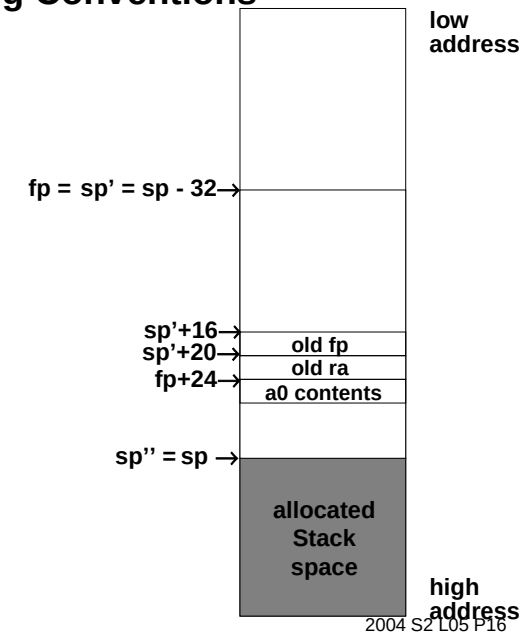
MIPS: Software conventions for Registers

0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
. . .		(callee can clobber)
15	t7	
16	s0	callee saves
. . .		(caller can clobber)
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	Pointer to global area
29	sp	Stack pointer
30	fp	frame pointer
31	ra	Return Address (HW)

Plus a 3-deep stack of mode bits.

MIPS / GCC Calling Conventions

```
func:
    addiu $sp, $sp, -32
    sw    $ra, 20($sp)
    sw    $fp, 16($sp)
    add   $fp, $sp, $0
. . .
    sw    $a0, 24($fp)
...
    lw    $31, 20($fp)
    lw    $fp, 16($fp)
    addiu       $sp, $fp, 32
    jr    $31
```



Details of the MIPS instruction set

- Register zero always has the value zero (even if you try to write it)
- Branch/jump and link put the return addr. PC+4 into the link register (R31)
- All instructions change all 32 bits of the destination register (including lui, lb, lh) and all read all 32 bits of sources (add, sub, and, or, …)
- Immediate arithmetic and logical instructions are extended as follows:
  - logical immediates ops are zero extended to 32 bits
  - arithmetic immediates ops are sign extended to 32 bits (including addu)
- The data loaded by the instructions lb and lh are extended as follows:
  - lbu, lhu are zero extended
  - lb, lh are sign extended
- Overflow can occur in these arithmetic and logical instructions:
  - add, sub, addi
  - it cannot occur in addu, subu, addiu, and, or, xor, nor, shifts, mult, multu, div, divu

Summary: Instruction set design (MIPS)

- Use general purpose registers with a load-store architecture: YES
- Provide at least 16 general purpose registers plus separate floating-point registers: 31 GPR & 32 FPR
- Support basic addressing modes: displacement (with an address offset size of 12 to 16 bits), immediate (size 8 to 16 bits), and register deferred; : YES: 16 bits for immediate, displacement (disp=0 => register deferred)
- All addressing modes apply to all data transfer instructions : YES
- Use fixed instruction encoding if interested in performance and use variable instruction encoding if interested in code size : Fixed
- Support these data sizes and types: 8-bit, 16-bit, 32-bit integers and 32-bit and 64-bit IEEE 754 floating point numbers: YES
- Support these simple instructions, since they will dominate the number of instructions executed: load, store, add, subtract, move register-register, and, shift, compare equal, compare not equal, branch (with a PC-relative address at least 8-bits long), jump, call, and return: YES, 16b
- Aim for a minimalist instruction set: YES

Summary: Evaluating Instruction Sets?

Design-time metrics:

° Can it be implemented, in how long, at what cost?

° Can it be programmed? Ease of compilation?

Static Metrics:

° How many bytes does the program occupy in memory?

Dynamic Metrics:

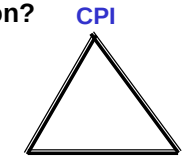
° How many instructions are executed?

° How many bytes does the processor fetch to execute the program?

° How many clocks are required per instruction?

° How "lean" a clock is practical?

Best Metric: Time to execute the program!



NOTE: this depends on instructions set, processor organization, and compilation techniques.

Exercise 1

Add comments to the following MIPS code and describe in one sentence what it computes. Assume that $a0 is used for the input and initially contains n, a positive integer. Assume that $v0 is used for the output.

```
begin:   addi $t0, $zero, 0
         addi $t1, $zero, 1
loop:    slt  $t2, $a0, $t1
         bne  $t2, $zero, finish
         add  $t0, $t0, $t1
         addi $t1, $t1, 2
         j    loop
finish:  add  $v0, $t0, $zero
```

Exercise 1

Add comments to the following MIPS code and describe in one sentence what it computes. Assume that $a0 is used for the input and initially contains n, a positive integer. Assume that $v0 is used for the output.

```
begin:   addi $t0, $zero, 0      # set $t0 = 0
         addi $t1, $zero, 1      # set $t1 = 1
loop:    slt  $t2, $a0, $t1      # while (n >= $t1)
         bne  $t2, $zero, finish #
         add  $t0, $t0, $t1      #   add $t1 to $t0
         addi $t1, $t1, 2        #   add 2 to $t1
         j    loop               # end while
finish:  add  $v0, $t0, $zero    # return $t0
```


Exercise 1

Add comments to the following MIPS code and describe in one sentence what it computes. Assume that $a0 is used for the input and initially contains n, a positive integer. Assume that $v0 is used for the output.

```
begin:   addi $t0, $zero, 0
         addi $t1, $zero, 1
loop:    slt  $t2, $a0, $t1
         bne  $t2, $zero, finish
         add  $t0, $t0, $t1
         addi $t1, $t1, 2
         j    loop
finish:  add  $v0, $t0, $zero
```

OP	RS	RT	IMMED
OP	RS	RT	IMMED
OP	RS	RT	RD
OP	RS	RT	ADDRESS
OP	RS	RT	RD
OP	RS	RT	IMMED
OP	ADDRESS		
OP	RS	RT	RD

Part 2: ALU Design

The Design Process

"To Design Is To Represent"

Design activity yields description/representation of an object

- **Traditional craftsman does not distinguish between the conceptualization and the artifact**
- **Separation comes about because of *complexity***
- **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, . . .**

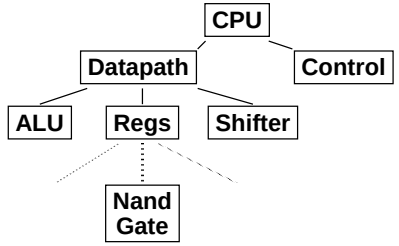
Design Process (cont.)

Design Finishes As Assembly

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

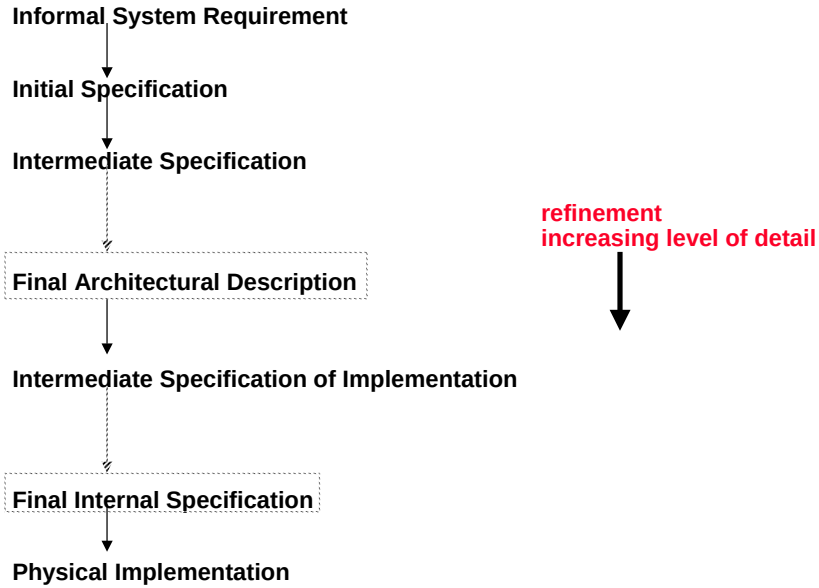
-- Top Down *decomposition* of complex functions (behaviors) into more primitive functions

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

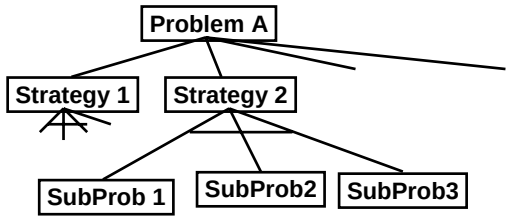


Design is a "creative process," not a simple method

Design Refinement



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

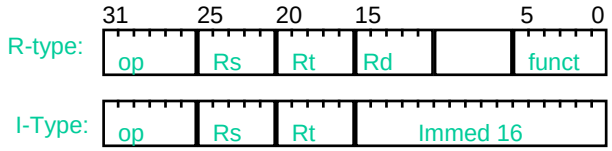
Problem: Design a “fast” ALU for the MIPS ISA

- 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, AddI, AddIU
  - => 2's complement adder/sub with overflow detection
- And, Or, AndI, OrI, 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
- ALU from P&H book Chapter 4 supports these ops

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 arith generate overflow, no carry

Design Trick: divide & conquer

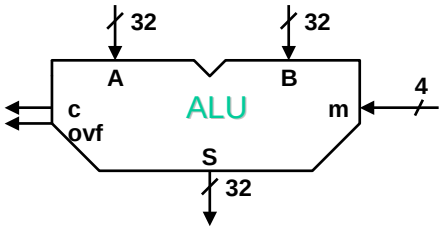
- Break the problem into simpler problems, solve them and glue together the solution
- Example: assume the immediates have been taken care of before the ALU
  - 10 operations (4 bits)

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
outputs: 32-bit result S, 1-bit carry, 1 bit overflow
operations: add, addu, sub, subu, and, or, xor, nor, slt, sltU

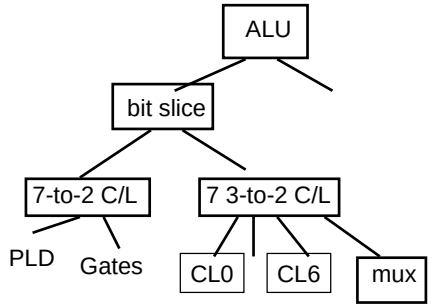
(2) Block Diagram (schematic symbol, VHDL entity)



Behavioral Representation: VHDL

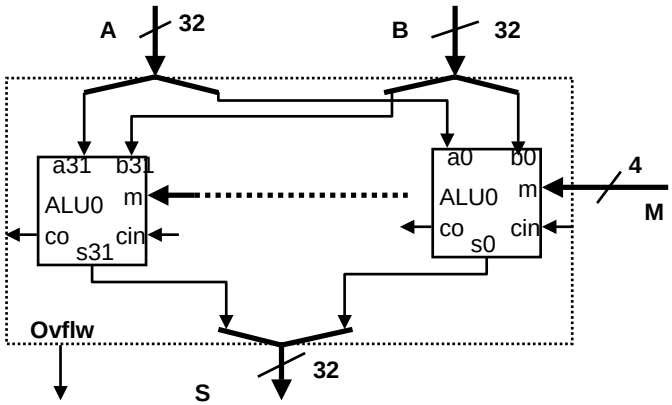
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

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

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

      S <= CONV_STD_LOGIC(CONV_INTEGER(A) +
                          CONV_INTEGER(B));
```


Design Decisions



- **Simple bit-slice**
  - **big combinational problem**
  - **many little combinational problems**
  - **partition into 2-step problem**
- **Bit slice with carry look-ahead**
- **. . .**

Refined Diagram: bit-slice ALU



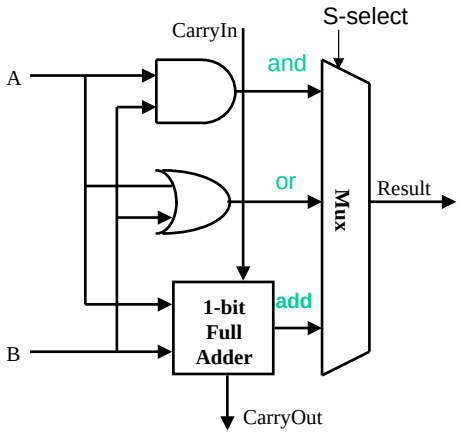
7-to-2 Combinational Logic

- **start turning the crank . . .**

	Function	Inputs							Outputs		K-Map
		M0	M1	M2	M3	A	B	Cin	S	Cout	
0	add	0	0	0	0	0	0	0	0	0	
127											

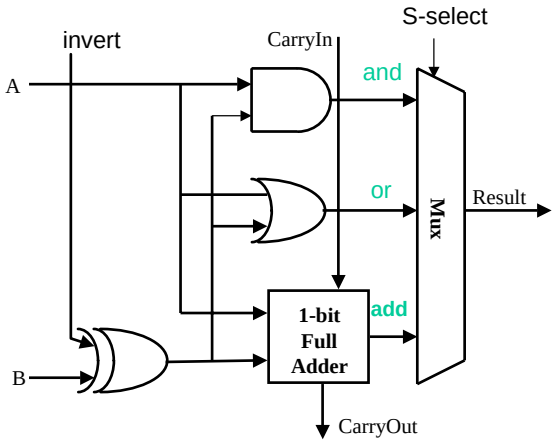
Seven plus a MUX ?

- **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**



Additional operations

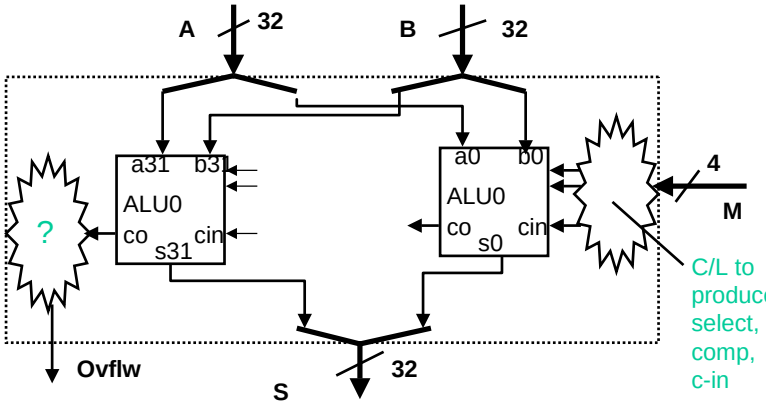
- **A - B = A + (– B)**
  - **form two complement by invert and add one**



Set-less-than? – left as an exercise

Revised Diagram

- **LSB and MSB need to do a little extra**

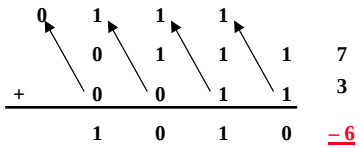


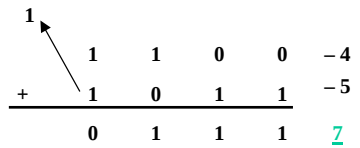
Overflow

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

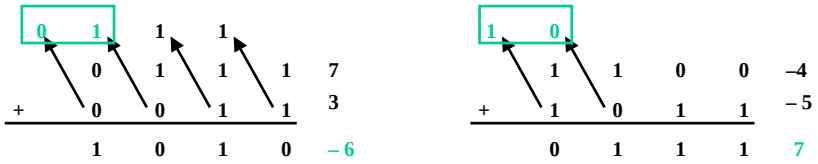
- Examples: 7 + 3 = 10 but ...
- - 4 - 5 = - 9 but ...





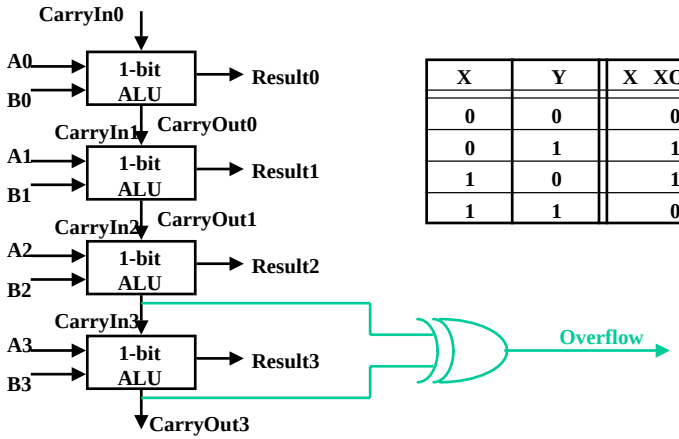
Overflow Detection

- Overflow: the result is too large (or too small) to represent properly
  - 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

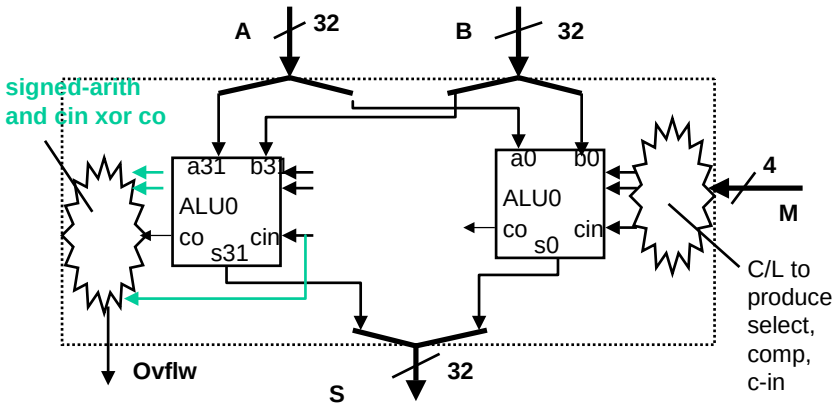
- Carry into MSB ⊕ Carry out of MSB
  - For a N-bit ALU: Overflow = CarryIn[N - 1] XOR CarryOut[N - 1]



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

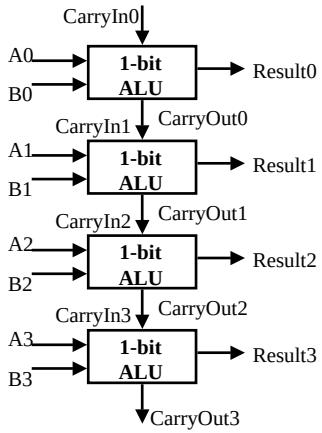
More Revised Diagram

- LSB and MSB need to do a little extra



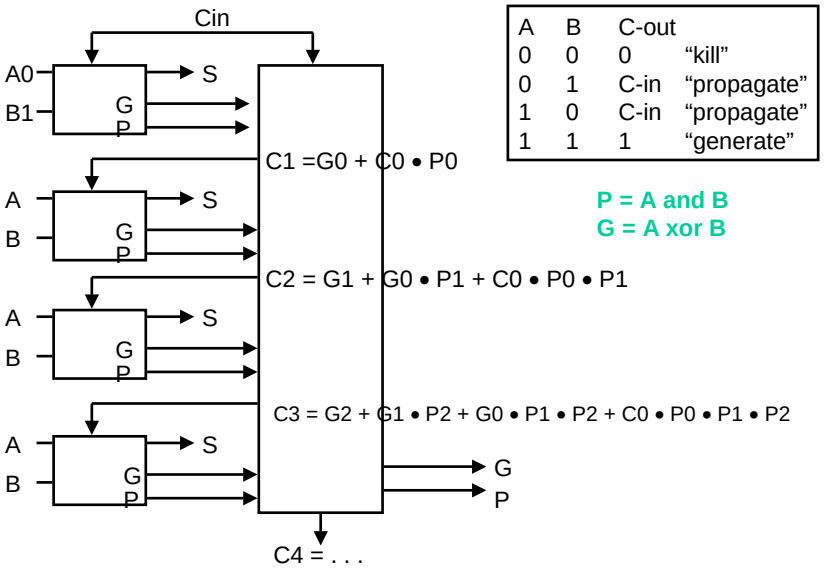
But What about Performance?

- Critical Path of n-bit Ripple-carry adder is n*CP



Design Trick: throw hardware at it

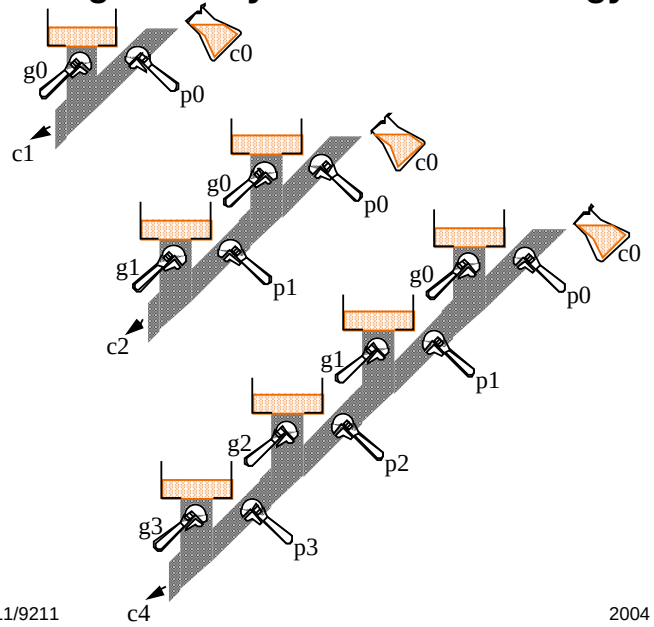
Carry Look Ahead (Design trick: peek)



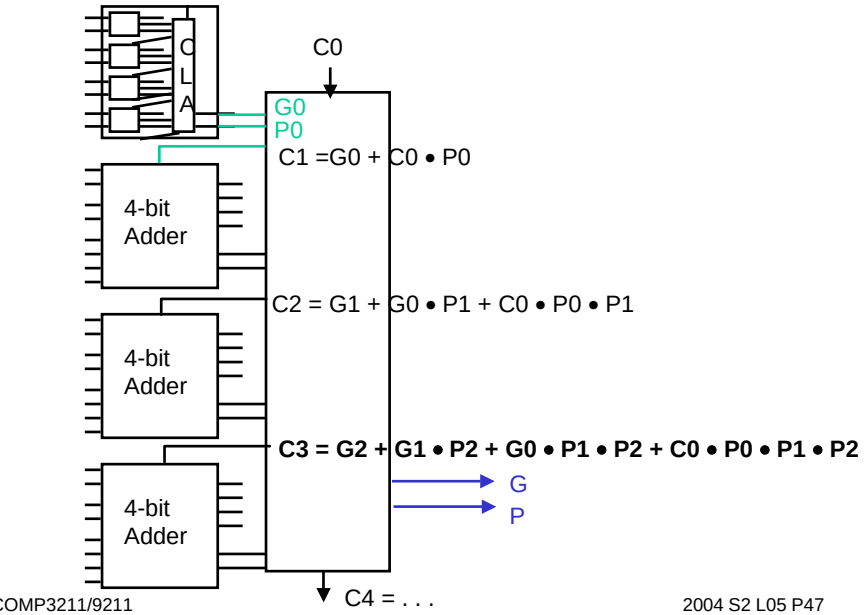
A	B	C-out	
0	0	0	"kill"
0	1	C-in	"propagate"
1	0	C-in	"propagate"
1	1	1	"generate"

P = A and B
G = A xor B

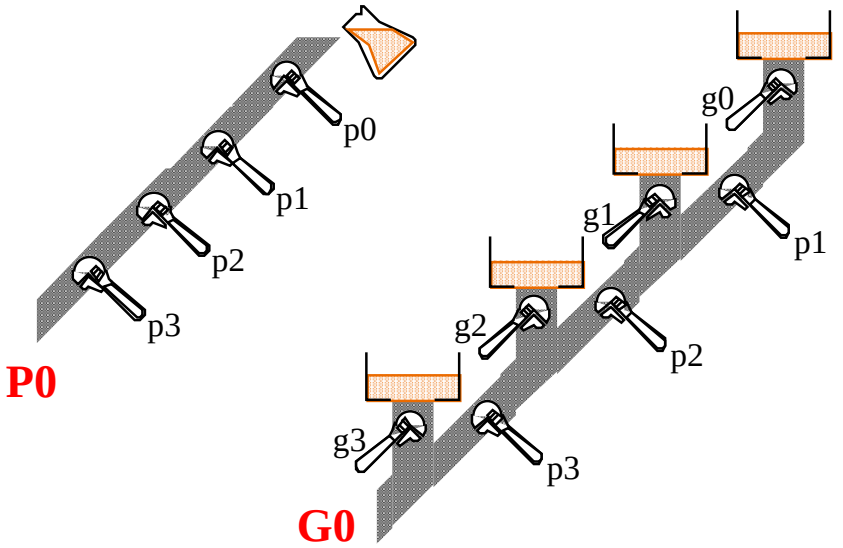
Plumbing as Carry Lookahead Analogy



Cascaded Carry Look-ahead (16-bit): Abstraction

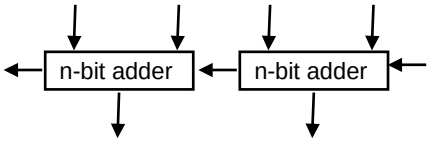


2nd level Carry, Propagate as Plumbing

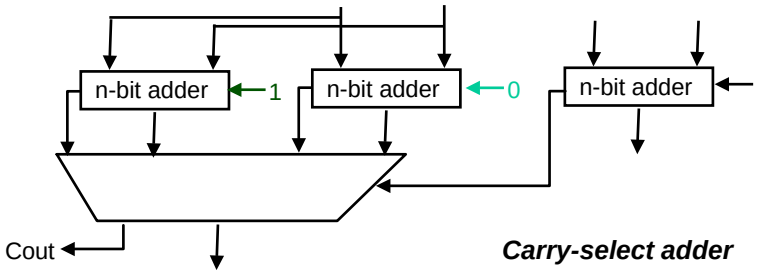


Design Trick: Guess

CP(2n) = 2*CP(n)

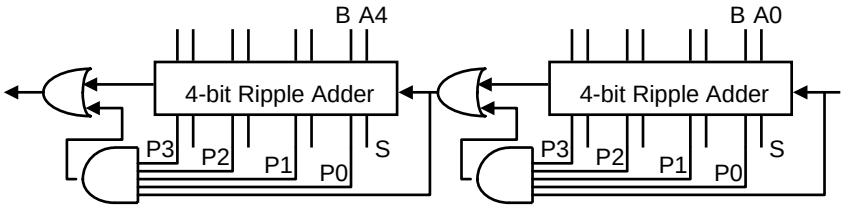


CP(2n) = CP(n) + CP(mux)



Carry-select adder

Carry Skip Adder: reduce worst case delay



Just speed up the slowest case for each block

Additional MIPS ALU requirements

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

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), examine and correct shortcomings.**
- **Formulate High-Level Alternatives (e.g., carry select)**
  - **Articulate many strategies to "keep in mind" while pursuing any one approach.**
- **Work on the Things you Know How to Do**
  - **If you wait until you know everything before you start, you will never get anything done.**
  - **The unknown will become “obvious” as you make progress.**

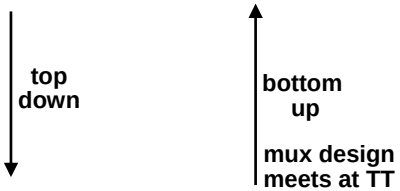
Summary of the Design Process

Hierarchical Design to manage complexity

Top Down vs. Bottom Up vs. Successive Refinement

Importance of Design Representations:

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



- **Other Descriptions: state diagrams, timing diagrams, reg xfer, . . .**

Optimization Criteria: