CS M151B:
Computer Systems Architecture

Week 2

Piazza Policy

- If you have a question, your first recourse should be Piazza.
- If you know the answer to a Piazza question, please respond.
- A TA will answer the unanswered questions once every day, but we encourage you to discuss/participate/answer.
- Will there be extra credit?

No

CISC

- CISC: Complex Instruction Set Computing
  - Developed organically, driven by memory constraints.
  - More compact code
  - ISA closer to high level language
  - Fewer instructions executed, more instructions defined.

CISC: Faster?

- Fewer instructions, but instructions become very complex, possibly making execution slower.
- ET = IC * CPI * CT
- Decreasing IC may result in a disproportionately larger increase of CPI and CT.

Other CISC Properties

- Variable Instruction Length
- 2 address instructions for simple, small instructions:
  - addl %eax %ebx
- But also crazy memory addressing to reduce no. instructions:
  - movl 16(%eax, %ebx, 8) %edx
- Retroactively named CISC as we moved towards RISC

RISC

- Memory was no longer a problem.
- The complexity of CISC instructions made execution slower.
- Move towards parallelism to speed up.
  - Instruction level parallelism (multiple instructions in same thread done simultaneously by breaking apart actions of instruction)
  - Difficult with super complicated and varying instructions.

RISC

- Fewer instructions defined, more instructions executed.
- Fixed instruction length
- More regularity
- Simple addressing
- More registers?
- Pipelining!

CISC vs. RISC

- These philosophies are merging together.
- Some CISC ISA's now implement complex instructions by breaking them down into the smaller instructions
- Turns CISC to RISC from the microarchitecture perspective.
- Some RISC ISA's may define pseudo instructions: complicated instructions that assemblers will just break into real instructions
- Turns RISC to CISC from assembly perspective.

MIPS, aka unlearn x86

- Simpler than AT&T x86, which was covered in CS33.
- Similar enough in core functionality to be familiar.
- Different enough to be obnoxious.

MIPS: Register to Register

- add $rd, $rs, $rt
- rs <=> source
- rt <=> tsource (?)
- rd <=> destination
- Destination is usually the first operand.
- *Usually*.

Enter: Memory Insns

- lw $rt, offset($rs)
- Load word FROM memory address $rs + offset into $rt.
- $rt or first operand is “destination” register
- What does store look like?

MIPS: Memory

- lw $rt, offset($rs)
- sw $rt, offset($rs)
- Store word at $rt TO memory address offset + $rs.
- Note: offset must be an immediate. Why?

MIPS: Branch

- beq $rs, $rt, offset
- Branch if equal: If $rs == $rt, then branch to PC + 4 + 4*offset
- Note: offset also has to be immediate
- Also, bne or branch if not equal.
- More on Branches and Jumps later.

MIPS Instruction Encodings:

- Format:
- [6 bits][5 bits][5 bits][5 bits][5 bits][6 bits]
- opcode rs rt rd shift funct
- Used for register to register operations such as add, etc.
- One opcode indicates that it is an R-Type insns, funct specifies which insn.
- Why 5 bits for registers?
- Why 5 bits for shift amount?

- MIPS has 32 registers. To index into them, you need 5 bits ($2^5 = 32$)
- Having 5 bits as a shift amount means you can shift up to 31 times. Would you ever need to shift more than 31 times if the word size is 32 bits?

I-Type

- Format:
- [6 bits][5 bits][5 bits][16 bits]
- opcode rs rt imm / addr
- Used for branches and memory access.

I-Type: Mem

- Format:
- [6 bits][5 bits][5 bits][16 bits]
- opcode rs rt imm / addr
- lw $rt, imm($rs)
- $rt = MEM[$rs + imm]
- Imm can be at most $2^{16} - 1$. How do you access memory with a greater offset?

I-Type: Mem

- Add offset to memory base location via add instruction and registers.
- This means that a single memory operation can take up to two operations.
- The simplicity of the fixed length encoding formats has increased the IC.
- Is it worth extending the entire ISA just to save the occasional instruction?

I-Type: Branch

- Format:
- [6 bits][5 bits][5 bits][16 bits]
- opcode rs rt imm / addr
- beq $rt, $rs, imm
- If $rt == $rs, go to PC + 4 + 4*imm
- Why + 4?
- Why imm*4?

I-Type Branch

- Why + 4?
  - By convention. However, if the branch is not taken, then PC + 4 must be done anyway. This way, PC + 4 will happen both when the branch is and isn't taken. Maybe helps in branch prediction
- Why Imm*4?
  - Recall that memory is byte addressable but instructions are every four bytes.
  - If we branched PC + 4 + imm, then we could only branch by a distance of 2^{15} and any imm that ends in 01, 10, 11 would be invalid.
  - Why don't we do this for memory instructions?

J-Type

- Format:
- [6 bits][26 bits]
- opcode addr
- For j, jal, but NOT jr. Why?
- j <addr>
- PC = [PC + 4]$_{31:28}$: 4*addr
- How do you jump farther?

Question 1

- R-Type instructions all have an opcode of 000000 with different funct values. Why?

Question 1

- R-Type instructions all have an opcode of 000000 with different funct values. Why?
  - Otherwise you'd have to share 64 different opcodes among all instruction types
  - This is feasible since R-Type instructions only really need to specify registers. The extra space is used for funct and shamt

Question 2

- How do you branch farther than $4*(2^{15} - 1)$ or $-4*2^{15}$?

Question 2

- How do you branch farther?
  - beq $s0,$s1, L1
  - ↓
  - bne $s0,$s1, L2
  - j L1
  - L2: …

Question 3

- How do you jump to a higher address than jump instructions allow?
  - Calculate address and store it into a register.
  - jr $rs
  - Would you need to?

Question 4

- Say we want to extend MIPS to include 64 registers.
- What direct ramifications to IC, CPI, and CT?

Question 4

- More registers means less loading and storing memory. This could lead to fewer instructions and lower CPI since loading and storing is expensive.
- Physically, more registers means slower access to register file, may increase CT (but I doubt it).
- What ramifications to instruction encoding?

Question 4

- R-Type
- [op (?)][rs(6)][rt(6)][rd(6)][SA(?)][funct(?)]

Question 4

- Steal from opcode:
- [op (3)][rs(6)][rt(6)][rd(6)][SA(5)][funct(6)]

Question 4

- Steal from opcode:
- [op (3)][rs(6)][rt(6)][rd(6)][SA(5)][funct(6)]
- Now any opcode of 000XXX must be interpreted as an R-Type. This affects the number of I-Type and J-Type instructions (7 fewer total instructions).
- Fewer defined instructions may mean more executed instructions, increasing IC.

Question 4

- Steal from SA:
- [op (6)][rs(6)][rt(6)][rd(6)][SA(2)][funct(6)]

Question 4

- Steal from SA:
- [op (6)][rs(6)][rt(6)][rd(6)][SA(2)][funct(6)]
- Now, we can only shift up to 3 times. That's not enough.
- Define shift some other way
  - May make ISA less regular/consistent.
- Perform multiple shifts when necessary.
  - Increase IC

Question 4

- Steal from funct:
- [op (6)][rs(6)][rt(6)][rd(6)][SA(5)][funct(3)]

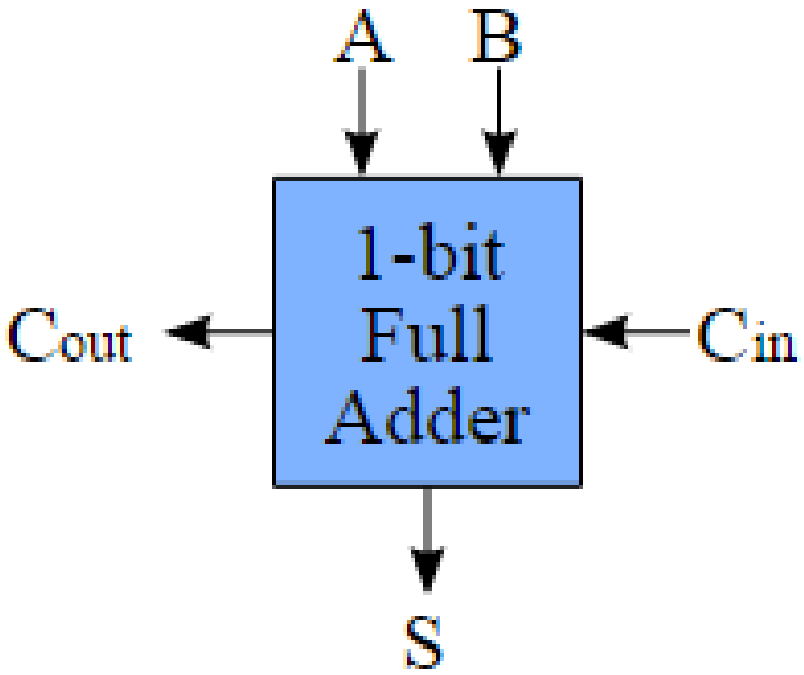
Question 4

- Steal from funct:
- [op (6)][rs(6)][rt(6)][rd(6)][SA(5)][funct(3)]
- Now there are only 8 R-Type instructions we can do.
- The best solution would probably be a stealing from a combination of opcode, shamt, and funct.
- Or perhaps the best solution is to extend the ISA and make it less regular?

Computer Arithmetic

- Dealing with 2's complement numbers
- Desired functionality:
  - AND
  - OR
  - NOR
  - Addition
  - Subtraction
  - Set less than

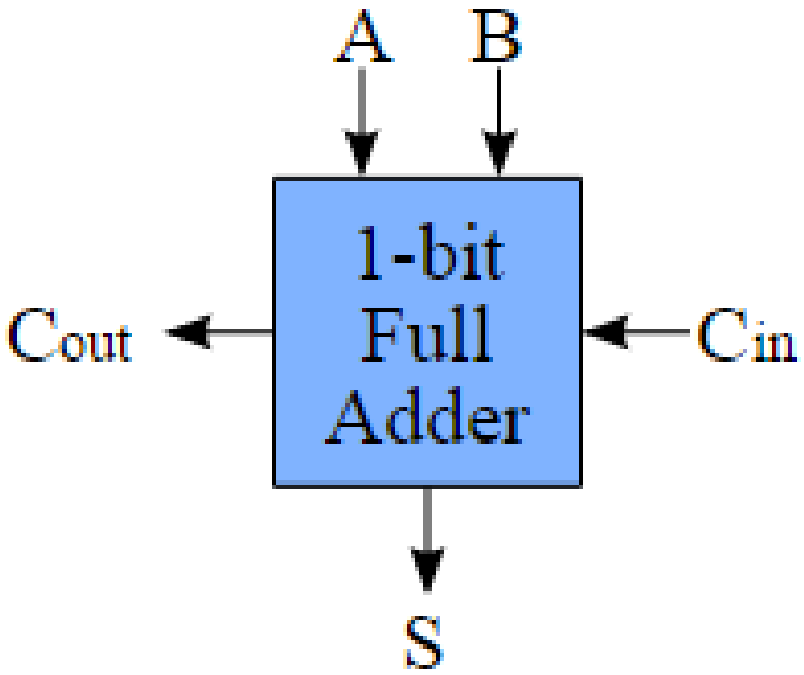
1-Bit Full Adder



- Inputs A and B
- Carry in: Cin
- Output: S
- Carry out: Cout

Picture courtesy of Professor Wikipedia

1-Bit Full Adder



- S = (!A & !B & Cin) |
  (!A & B & !Cin) |
  (A & !B & !Cin) |
  (A & B & Cin)
- S = (A ^ B) ^ Cin
- Which is better?

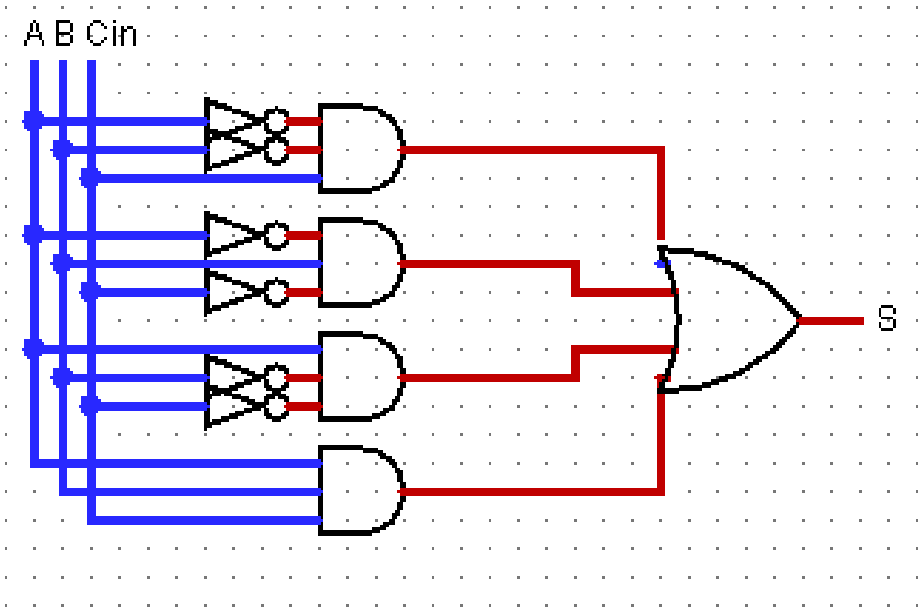
Picture courtesy of Professor Wikipedia

Measuring Latency

- Latency through a gate = fan in * T where T is some magic constant.
- This is rudimentary and not entirely accurate, but will be a fair guide for latency.

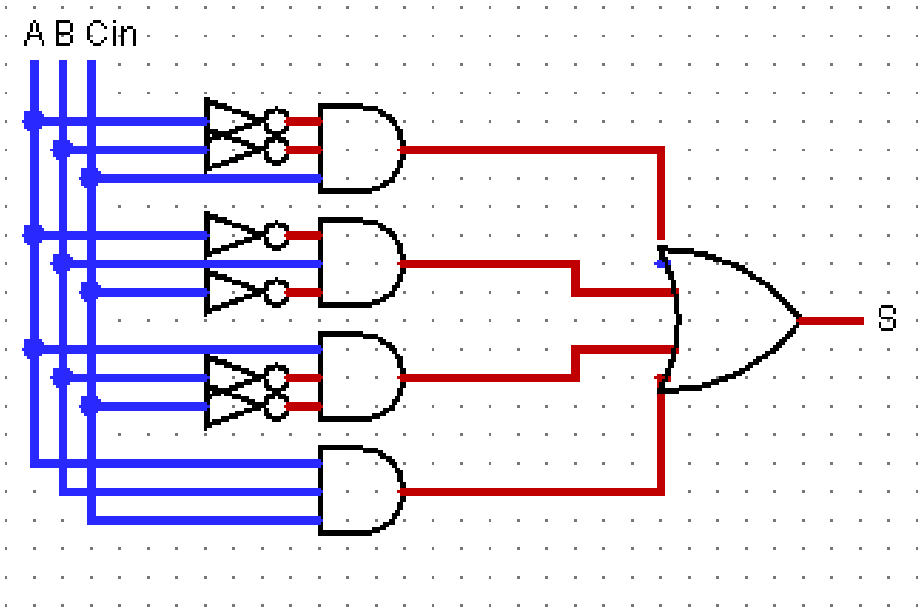
Version 1

- Latency?

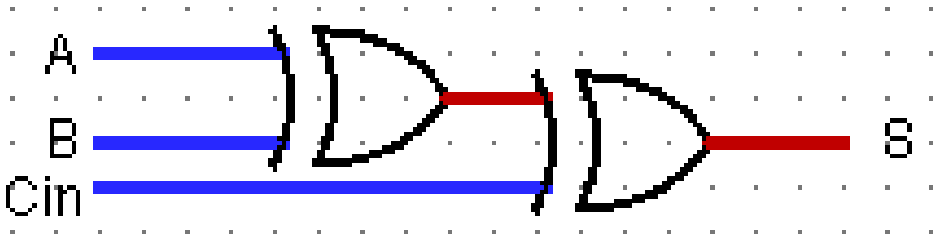


Version 1

- L = 1T + 3T + 4T = 8T

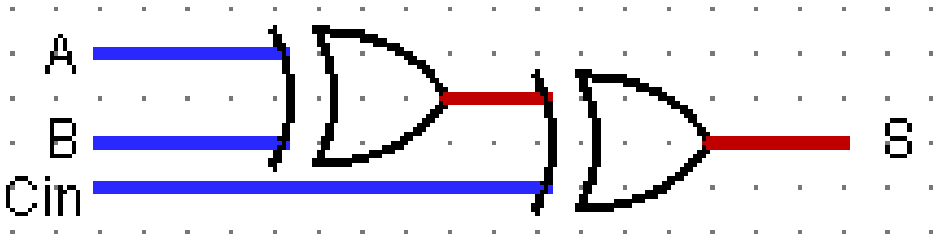


Version 2



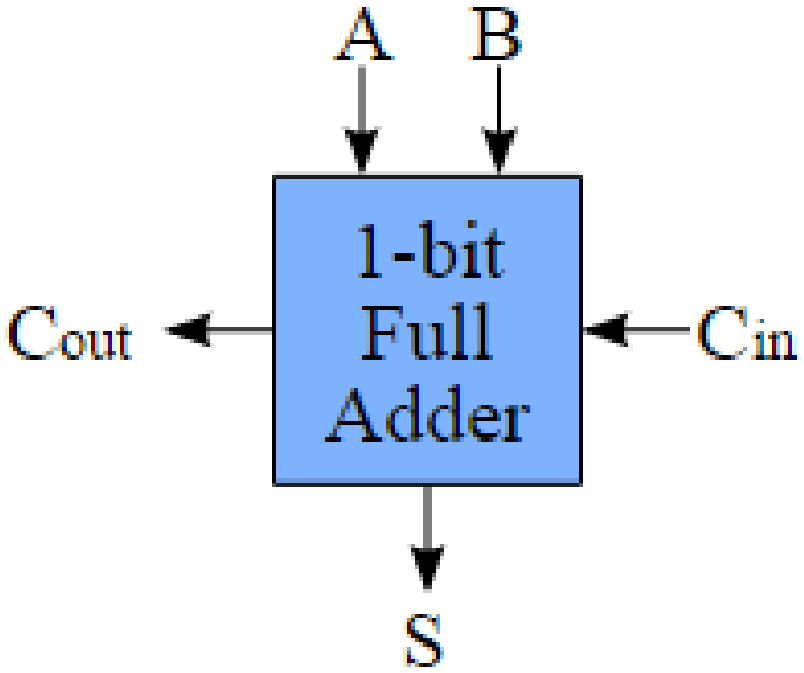
- Latency?

Version 2



- L = 2T + 2T = 4T

1-Bit Full Adder

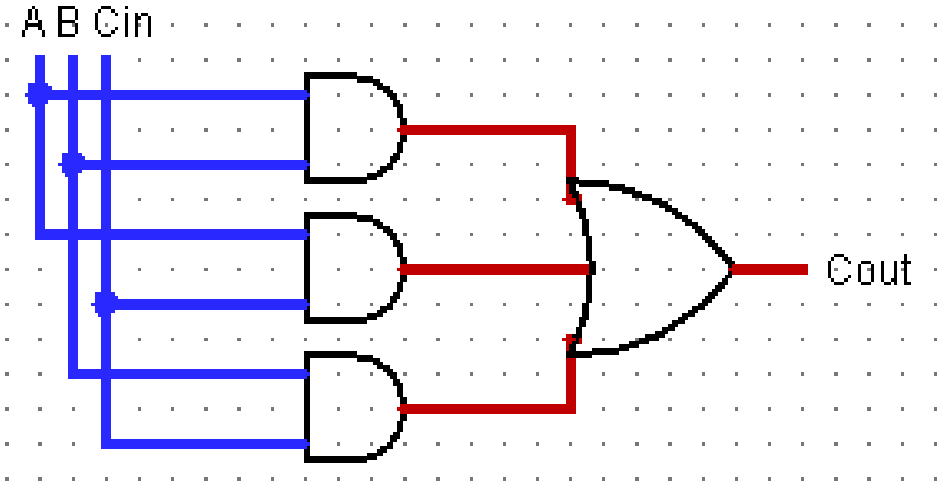


- Cout =
  (A & B) | (A & Cin) | (B & Cin)
- Cout = ((A ^ B) & Cin) | (A & B)
- Which is better?

Picture courtesy of Doctor Wikipedia

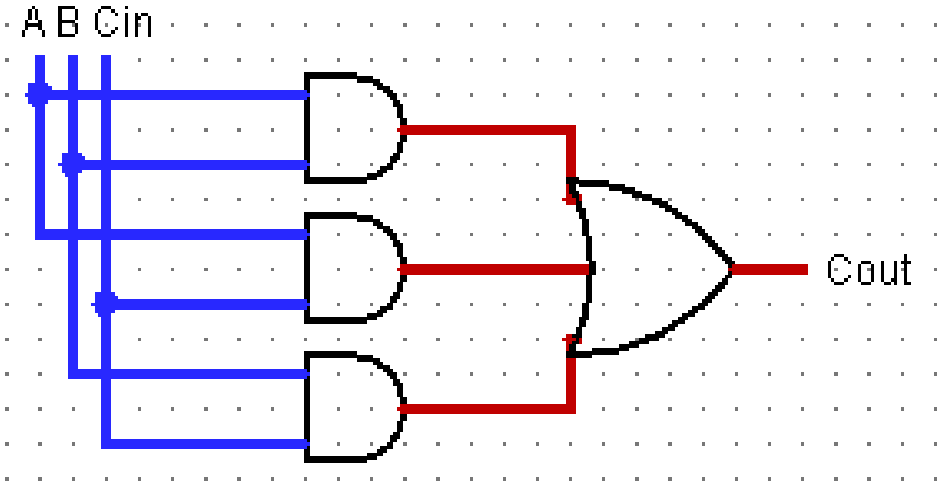
Version 1

- Latency?



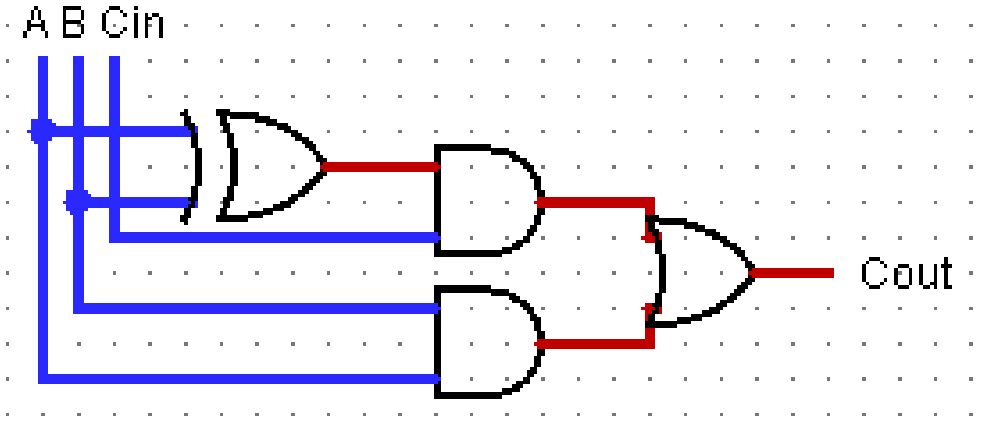
Version 1

- L = 2T + 3T = 5T

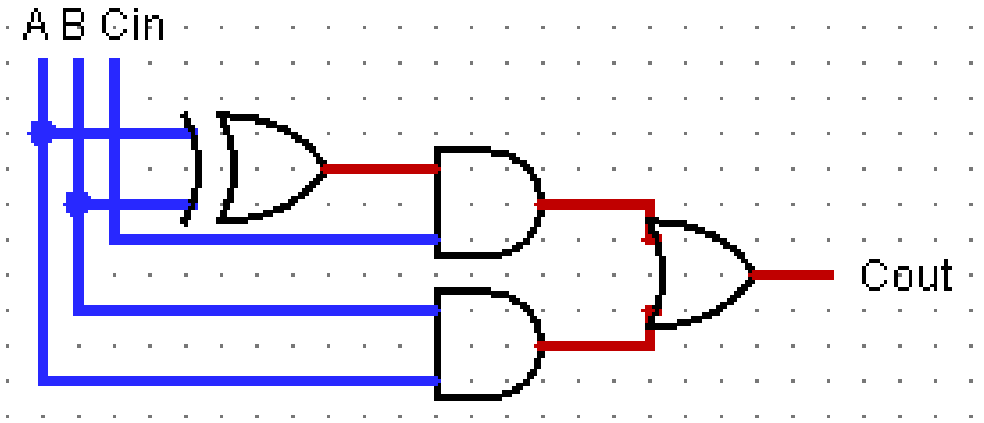


Version 2

- Latency?

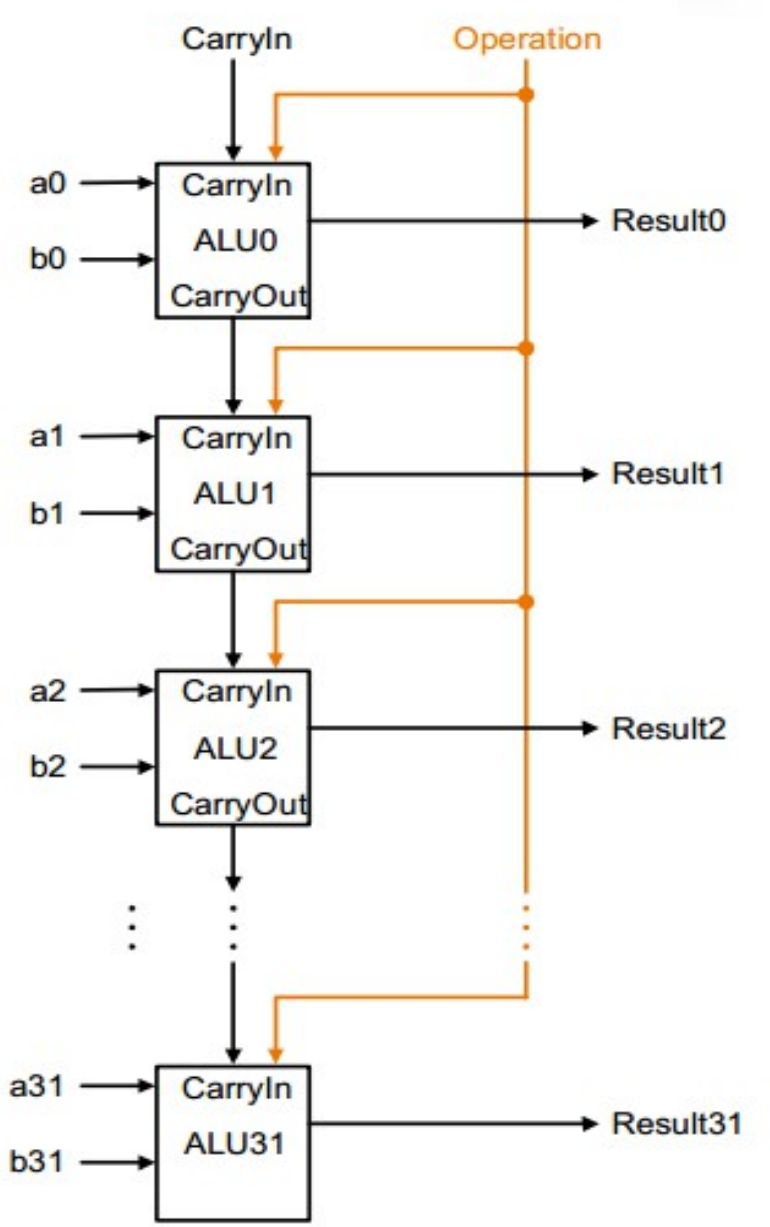


Version 2



- L = 2T + 2T + 2T = 6T
- Using this sort of latency measure, would it be better to do ((A & B) & C) with two 2-Input AND gates in series or (A & B & C) with one 3-input AND gate?

Ripple Carry Adder



- Chain 1-bit full adders together.
- Cout of Adder N – 1 leads into Cin of Adder N.
- What's the minimum clock cycle we must use to be able to use this?

Terminology

- Cn : Cin into unit n
- Cout_n : Cout of unit n
- Cout_n = Cn+1

Latency

- Critical path is due to propagation of carries.
- S = (A ^ B) ^ Cin
- Cout = (A & B) | (A & Cin) | (B & Cin)
- Assume Cin0 is ready at time 0.
- What is latency of S0?
- What is latency of Cout0?

Latency

- S0 = 4T
- Cout0 = 5T
- What is the latency of S1?

Latency

- S0 = 4T
- Cout0 = 5T
- What is the latency of S1?
- S1 = (A1 ^ B1) ^ Cout0
- Cout0 is ready at time 5T, but A and B are ready at time 0. Since A1 ^ B1 (latency of 2T) is done in parallel with computation of Cout0 (latency of 5T), the Cout0 is part of critical path.
- S1 is ready at 5T + 2T (final xor) = 7T
- In general, Sn = ?

Latency

- In general...
- Sn = Cout_n-1 + 2T if n > 0
- Sn = Cn + 2T
- What is latency of Cout1?

Latency

- In general...
- Sn = Cout_n-1 + 2 if n > 0
- What is latency of Cout1?
- Cout1 = (A1 & B1) | (A1 & Cout0) | (B1 & Cout0)
- A and B are ready at time 0, Cout0 is ready at 5T.
- Cout1 = 5T (time for Cout0) + 2T (AND-ing A/B with Cout0) + 3T (Or-ing all inputs together) = 10T

Latency

- In general, what is Cout_n?

Latency

- In general, what is Cout_n?
- Cout_n = Cout_n-1 + 5T
- Cout_n = (n + 1) * 5T
- What is S31?
  - S31 = Cout_30 + 2T = 157T
- What is Cout_31?
  - Cout_31 = (32) * 5T = 160T
- 160 T's? That's a lot of T's.

Carry Lookahead

- Sacrifice space for speed

Carry Lookahead

- When is S0 ready?
  - S0 = 4T
- When is Cout_0 ready?
  - Cout_0 = G0 + Cin_0 * P0 = 4T
  - G0 is A0 * B0 so G0 is ready at 2T
  - P0 is A0 ^ B0 so P0 is ready at 2T
  - Cin_0 is ready at time 0
  - Cout_0 = 2T (P0) + 2T(Cin_0 * P0) + 2T(G0 + …) = 6T.
  - In slides Cout_0 is same as C1 (Cin_1).

Carry Lookahead

- When is S1 ready?
  - S1 = Cout_0 + 2T = 8T
- When is Cout_1 (or C2) ready?
  - Cout_1 = G1 + Cout_0 * P1
  - Cout_1 = G1 + (G0 + Cin_0 * P0) * P1
  - Cout_1 = G1 + G0 * P1 + Cin_0 * P0 * P1
- Do we need to wait for Cout_0 (or C1)? That takes 6T to be ready!

Carry Lookahead

- Recall, trading space for speed.
- Cout_1 = G1 + G0 * P1 + Cin_0 * P0 * P1
- This breaks down the equation so that it only relies on G's and P's which are ready at time 2T.
- Cout_1 = 2T (G's and P's in parallel) + 3T (Cin_0 * P0 * P1) + 3T (sum of terms) = 8T

Carry Lookahead

- Cout_1 = 2T (G's and P's in parallel) + 3T (Cin_0 * P0 * P1) + 3T (sum of terms) = 8T
- Generally speaking:
  - Cout_n (or Cn+1 by slide's notation) = 2T (G's and P's in parallel) + (N+2)T (Cin_0 * P0 * P1 *... PN) + (N+2)T (sum of terms) = (2N + 6) T
- Cout_31 = 68 T
- Way better... but...

Carry Lookahead

- The very last carry is going to be crazy, requiring two 32 fan in gates. That's not very practical.
- Maybe we sacrificed too much in the realm of space.
- How do we make it better?

Partial Carry Lookahead

- Chain smaller Carry Lookahead adders together and accept the loss of speed. Consider Carry Lookahead units of 4 bits.
- S3 = 12T
- Cout_3 or C4 = 12T
- Now we link to new Carry Lookahead Unit. Instead of doing:
  - Cout_4 or C5 = G4 + G3*P4 + G2*P3*P4 + …
- We do:
  - Cout_4 or C5 = G4 + C4 * P4

Partial Carry Lookahead

- Recall Cout_3 or C4 = 12T
- Cout_4 or C5 = G4 + Cout_3 * P4
- Cout_4 or C5 = 12T (Cout_3) + 2T (Cout_3 * P4) + 2T (sum the terms together) = 16T
- Cout_5 or C6 = 18T
- Cout_6 or C7 = 20T
- Cout_7 or C8 = 22T
- Each Partial Carry Lookahead Unit adds 10 T per 4 bits.

End of

The Second Week

-Eight Weeks Remain-