- Moore's Law: # of transistors on chip doubles roughly every 2 years — 1971 - 2018
- Dennard Scaling: As transistors get smaller, their power density stays constant. This means as transistors double, power consumed stays constant. This broke down ~ 2008 b/c power leaks. Now, multi-core preferred

- 32-bit reg: reg has 32 bits of capacity
- 32-bit instr: machine code instr w/ 32 bits
- 32-bit address: mem address w/ 32 bits
- 32-bit OS: Computer w/ 32-bit addresses
- Word: generally 32 bits (Both here, 32-bit instr.)
- ARM: 64-bit registers, 32-bit addressing
  32 registers
  ARCHITECTURE

LC2K ISA: 32 bit

0. ADD RA RB R-Dest — R-Dest = RA + RB
1. NOR RA RB R-Dest — R-Dest = !(RA | RB)
2. LW RA RB #offset — RB = RA + offset
3. SW RA RB #offset — [RA + offset] = RB
4. BEQ RA RB Branch to PC + offset + 1 iff RA == RB (else inc PC)
5. JALR RA RB — 1. RB = PC + 1 2. PC = RA
6. HALT
7. NOOP

If A == B, then ++PC.

- 21-19: REG A
- 18-16: REG B
- 24-22: OPCODE
- 2-0: REG DEST
- 15-0: offset field
- ALL UNUSED BITS: 0 • 0x31: Least → Most Sig

- LC2K: 32-bit registers, 16-bit address, 8 reg

Special ARM Regs:
- X30: LR, return addy of subroutine
- X28: SP, stack pointer
- X29: FP, start of stack frame
- X31: X2R, always 0

Addressing Modes:
- Direct (mem address in instruction)
- Indirect (mem address in register)
- Base + displacement (reg + immediate)
  → PC + immediate

Data Sizes: depends on Architecture — # bits in address

Type	32-bit	64-bit
CHAR	1	1
SHORT	2	2
INT	4	4
FLOAT	4	4
DOUBLE	8	8
LONG	4	8
PTR	4	8

ARM/LEG:
- Byte-addressable
  — remember this when accessing arrays, PC+=4
- 9-bit signed immediate (-256 ≤ x ≤ 255) for offset
- Recall register is 64-bits, so 0-extend or sign-extend
- Double-word load depends on endianness of system
- Big endian: Most → Least

Two's Complement
- First bit is negative
  → $1011_2 = -2^3 + 2^1 + 2^0$
- Represents $[-2^{n-1}, 2^{n-1}-1]$
- Negate positive number:
  1. Flip all bits
  2. Add 1

Calling a Function
1. Pass arguments (registers and stack if necessary)
2. Save return address of caller (register)
3. Save register values
4. Jump to function, execute, jump to return address
5. Get return value (reg)
6. Restore reg values

- ARM: X0-X15 CALLER
  X19-X27 CALLEE
  X0-X3 ARGUMENTS
  X0 RETURN VAL

- Parameters of a function are analyzed as local vars in that func's analysis

Caller / Callee Save
- Each register is defined by ISA as caller-save or callee-save
- Compiler decides what type to use for certain function / value
- Caller-save: Every time function called, save necessary registers and restore after call.
- Callee-save: At start + end of a function, save/load all regs used during function call
- Caller-save advantage: Only needs saving if live across function call
- Callee-save advantage: Only one load/save per register
- Do not optimize for multiple function calls back-to-back

```
void main {
  int a, b, c, d;
  a = 0; b = 0;
  c = 0; d = 0;
  foo();
  d = a + b + c + d;
}

void foo {
  int e, f;
  e = 2;
  f = 3;
  bar();
  e = e + f;
}

void bar {
  int g, h, i, j;
  g = 0; h = 1;
  i = 2; j = 3;
  final();
  j = g + h + i;
}

void final {
  int y, z;
  y = 2;
  z = 3;
  z = y + z;
}
```

CALLER:	4	2	3	0 → 9
CALLEE:	0	2	4	2 → 8
SPLIT 3/3:	1/3	?	3/0	2/0
	1+0=1	2	3	0 → 6

- ALL OPTIMIZATIONS ARE FUNCTION-LOCAL!!!
- main() always uses 0 for callee save
- Remember to multiply in tally for callee for # of times func is called
- Leaf functions always use 0 for caller-save
- Treat recursive call as unknown func

Template: Conditional

```
if x == Y          1     CMP X Y
  // do A          2     B.EQ if
else               3     // do B
  // do B          4     B endif
                   5 if  //do A
                   6 endif
```

Template: Loop

```
i = 0;
while (i < 10)     1     MOV2 X1 #0
  //do A           2     MOV2 X2 #10
  ++i              3 start CMP X1 X2
                   4     B.GTE end
                   5     //do A
                   6     ADDI X1 X1 #1
                   7 end
```

Structs / Alignment:
- Address of variable aligned based on size — start address must be multiple of size
  - For arrays, use size of primitive in arr
- Size of overall struct MUST be multiple of largest) field — pad @ the end (primitive again)

Source Code to Execution:
1. Compile each .cpp to .as
2. Assemble each .as to .o
   → 2 passes to resolve addys
3. Link .o files, library files together into one executable (.out)
4. Load .out file and use DLL (dynamically linked libraries) to help execute

Doubles: 64 bits:
1 sign, 11 exp, 52 mantissa
+1023 exponent bias

.O File Structure (Recall 1 .cpp → 1 .o)

HEADER
TEXT
DATA
SYMBOL TABLE
RELOCATION TABLE
DEBUG INFO

- Header: sizes of other parts ONLY
- Text: Machine code
  Read-Only Segment
- Symbol Table:
  - Map symbols to addresses
  - strings → values (values can be addresses or consts)
  - lists unresolved labels
  - function calls + globals
  - statics also here for consistency
- Relocation Table:
  - Identifies instructions (by address) that need to have addresses updated

ADDRESS (mult. 4)	INSTR	SYMBOL
0	LDUR	#D

  - All instr. that load/store stuff in data segment (static, global, extern)
  - All things that reference variables / funcs that are undefined in symbol table
  - Refs to locally defined functions DO NOT go here (in curr. .cpp)
  - Types: STUR, LDUR, BL
- Symbol Table: DOES have local functions, but not local vars

- Where data goes:
  → Local variables: stack
  → Arguments: 4 in regs rest on stack
  → Dynamic Memory: heap
  → Globals: Data
  → static vars: Data
  → const. strings: Data
- Data segment DOES NOT CONTAIN Stack / Heap
- #defines do not go anywhere
- In symbol table, anything external is "U"-type (unknown)
- Initializations / declarations are handled @ compile-time and don't need to be in relocation table

Floating Point
- IEEE Standard:
  [sign][exp][mantissa]
  31 30-23 22-0
  (1) (8) (23)
  Mantissa ONLY stores beyond decimal point;
- Exponent is 127 based to always be non-negative
- Example: $1.0101 \cdot 2^3$ (5.5)
  Sign: 0 (positive)
  Exp: $3 + 127 = 130 = 10000010$
  Mantissa: 01010000...
  - When converting, remember to add / take away implicit 1! Also remember exp bias.
- 0 is all 0s in exp/mantissa

Multiply Floats
1. XOR sign bit
2. Add exponents (w/o bias!!)
3. Multiply significands (mantissa w/ implicit 1s included)
4. Combine it back!

- Example: 10.625×10
  → $10.625 = 1.0101101 \cdot 2^3$
  → $10 = 1.01 \cdot 2^3$
  → Sign: 0 XOR 0 = 0
  → exp: $3 + 3 + 127 = 133 \rightarrow 10000101$

```
  1.0101101
x 1.01
-----------
  1.0101101
+ 1.0101101
-----------
  1.0101010 01
```

$1.01010101 \cdot 2^6$
$64 + 32 + 8 + 2 = 106.25$

Add Floats
1. Shift mantissa of number w/ smaller exponent right (BRING BACK IMPLICIT 1!)
2. Add significands
3. Update exponent if overflowed beyond limit of post exp

Example:
→ A: 0 | 10000101 | 10010100000...
→ B: 0 | 10000010 | 10111000000...
→ Shift B 3 to the right and add:
  1 100 101001
+ 000 1011110
= 1 101 0010111
no overflow! exp stays @ 6 (A)
mantissa is

Finite State Machine
- S: {states}, |S| = k
- I: {inputs}, |I| = N
- O: {outputs}, |O| = M
- Transition function T: (S, I) → S
- Output Function P:
  S → O (Moore)
  (S, I) → O (Mealy)

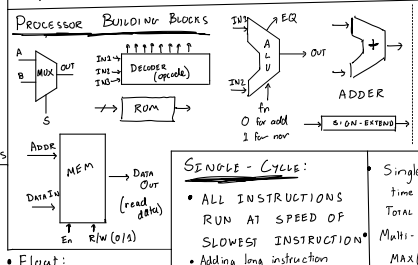
Implementing Combinational Logic:
1. Use gates to represent FSM
   → Can optimize # of gates used
   → Can be expensive / time consuming to make custom logic circuits
2. Use ROM (Read-Only Memory)
   → Cheap, fast to program
   → Size requirement grows exponentially w/ # of bits!!

- In Practice: Some combo of the two

ROM Size:
- Input bits: # of bits to represent states + # of possible inputs
  ($\lceil \log_2 k \rceil + n$)
- Output bits: # of bits to represent state + # of outputs
  ($\lceil \log_2 k \rceil + m$)
- $2^{\#\text{input}}$ entry bits
- # output memory
- Total: $2^{\#\text{input}} \times \#\text{output bits}$

Ranges for Nums
- n-bit signed int:
  $[-2^{n-1}, 2^{n-1} - 1]$
- n-bit unsigned int:
  $[0, 2^n - 1]$
- Float:
  $3.4 \cdot 10^{38}$ max val

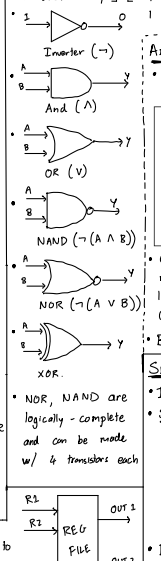
Processor Building Blocks



Single-Cycle:
- ALL INSTRUCTIONS RUN AT SPEED OF SLOWEST INSTRUCTION
  - Adding long instruction hurts performance
  - No benefit in making common case fast
  - Can't reuse parts of processor

- Single-Cycle cycle time:
  time to execute longest instr end-to-end
  TOTAL = [cycle time] × [num instr]
- Multi-Cycle cycle time:
  MAX (individual latency ops)
  TOTAL = [cycle time] × $\sum_{\text{instr}}$ [cycle/instr] [# instr]
  Generally, balancing delays ⇒ Multicycle performs better

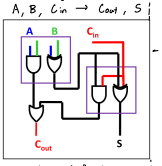
Combinational Logic:
- Transistor: First layer of abstraction,



- NOR, NAND are logically-complete and can be made w/ 4 transistors each

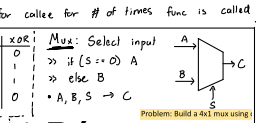
A	B	AND	OR	NAND	NOR	XOR
0	0	0	0	1	1	0
0	1	0	1	1	0	1
1	0	0	1	1	0	1
1	1	1	1	0	0	0

Addition
- For adding 2 bits: A, B, C_{in} → Cout, S



- Can "ripple" for multi-bit adds by going least → most sig and passing Cout as next Cin
- Basic gate of ALU

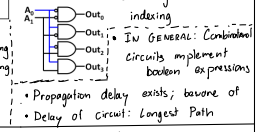
Mux: Select input
» if (S == 0) A
» else B
- A, B, S → C



Problem: Build a 4x1 mux using t

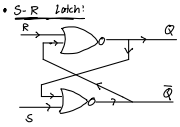
- Basic gate for data movement

Decoder:
- Input: n-bit binary number
- Output: 2^N bits; exactly 1 "on"
- Basic gate of indexing



- In General: Combinational circuits implement boolean expressions
- Propagation delay exists; because of
- Delay of circuit: Longest Path

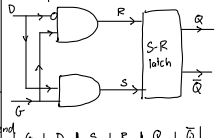
Sequential Logic: Give Memory to Circuits
- Idea: output depends both on input and current state.
- S-R Latch'



- Invariant: $Q = !\bar{Q}$
- Q, Q̄ retain values for 0,0 in

S	R	Q	Q̄	
0	0	Q	Q̄	Breaks invariant
0	1	0	1	
1	0	1	0	
1	1	0	0	INVALID! Leads to unstable future states if input changes

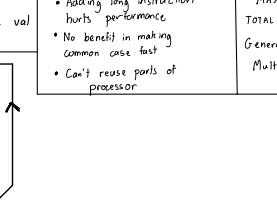
- D-Latch: Introduces safety mechanism to prevent 1-1 case in S-R



- G: Gating signal — change value iff G = 1
- D: Matters only if G = 1; the value you want for Q afterwards

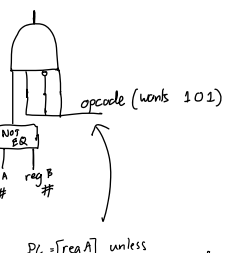
G	D	S	R	Q	Q̄
0	0	0	0	Q	Q̄
0	1	0	0	Q	Q̄
1	0	0	1	0	1
1	1	1	0	1	0

- New problem: latency time leads to error propagating in circuit.
  - Soln: Introduce Clock
    → only do things on rising edge
    → clock provides gating signal
    → clock period should be high enough to allow signal propagation from start → end of a latch



LC2K single-cycle datapath

JALR Extra Gate



update PC = [regA] unless regA# == regB#; then, PC = PC + 1

- Changes for JALR:
  1. PC + 1 adder sends output to MUX (2) as middle input
  2. now has 3 options (2 bits)
  2. [RegA] out from reg file now becomes middle input to leftmost MUX (that updates PC).

  Another AND gate is added for least sig bit of that leftmost MUX. Needs to be 1 on JALR as long as RegA ≠ RegB (reg #, not value).

3x8 DECODE — $2^3 \times 7$ ROM

	1 Writing to regB (0) or dest reg (1)?	2 Writing output of data memory (0) or ALU op (1)? (to REG FILE)	3 Write-enable register file? (1 to enable)	4 Compute offset + [regA] (0) or [regA] ⊕ [regB] (1)?	5 ALU add (0) or nor (1)	6 Data mem enable R/W (1)	7 Read Data Mem (0) or Write (1)
ADD	1	1	1	1	0	0	X
NOR	1	1	1	1	1	0	X
LW	0	0	1	0	0	1	0
SW	X	X	0	0	0	1	1
BEQ	X	X	0	1	X	0	X
JALR	0	01 (PC + 1) → write [PC + 1] to reg B	1	X	X	0	X

Latency Calculation:

	Read instr. (mem read)	Read register file	ALU OP	Mem Read	Mem Write	Write Reg
ADD	✓	✓	✓	X	X	✓
NOR	✓	✓	✓	X	X	✓
LW	✓	✓	✓	✓	X	✓
SW	✓	✓	✓	X	✓	X
BEQ	✓	✓	✓	X	X	X
JALR	✓	✓	X ? kinda	X	X	X

- Response Time (s): ExecTime = # instr × CPI × clock period
  cycles per instr.
- Throughput (#/s)

Things to Remember:

- Instructions are in Memory – when calculating single-cycle latency times, EVERY instr must read from mem @ start
- ADD/NOR/LW all have to access registers twice (R, W)
- Little endian: most sig @ HIGHEST memory address (remember that memory usually drawn flipped)
- Caller/Callee: Remember to multiply by # of calls for BOTH caller and callee calculations
- "How many times will the label be executed in this program?" means actual LABEL line, NOT branching to label
- When adding to datapath:
  - → Run thru control flow to ensure everything is there
  - → ADD CONTROL BITS @ BOTTOM AS NECESSARY
- ARM/LEG: Don't assume return regs/org regs – look @ code to figure this stuff out.

Symbol Table

Include
- Library/Extern funcs (U)
- Funcs in this file (T)
- Global Vars (D)
- Static Vars (D)
- Extern Vars (U)

Do Not Include
- #include, #define
- Local variables
- Definitions of classes Structs – only include when specific instance declared/initialized

Relocation Table

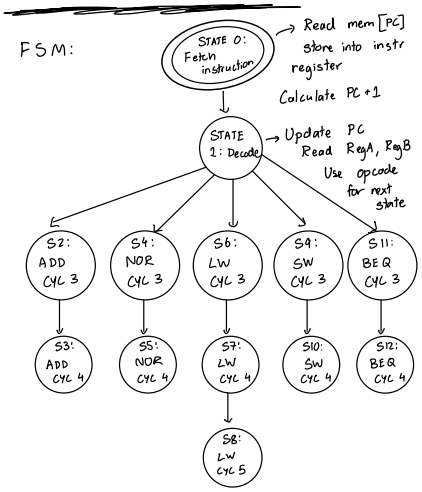
Include
- Extern/library function calls (BL)
- Global/static/extern variable load/store
  - recall ++ are BOTH L,S
  - LHS → STUR; RHS → LDUR

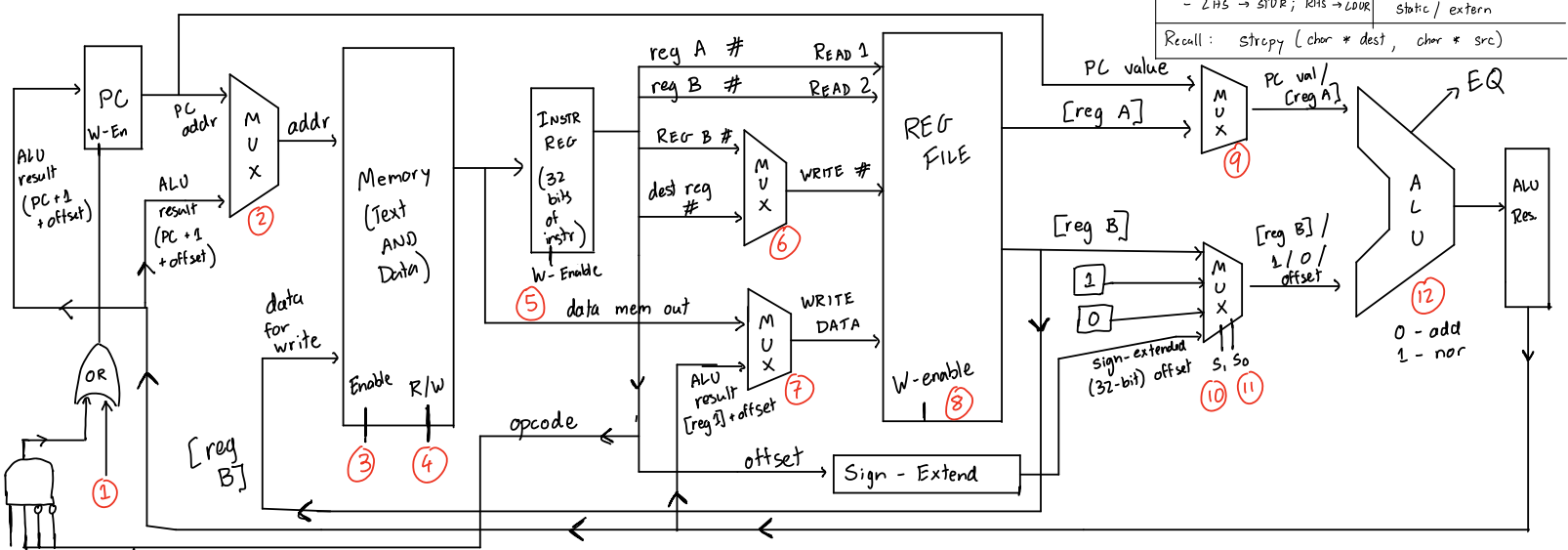
Do Not Include
- Locally-defined function calls
- Setting/getting local vars
- Declaration/declaration + initialization of global/static/extern

Recall: Strcpy (char * dest, char * src)

LC2K Multicycle Datapath

FSM:





① Are we writing to the PC this time? (1-Yes; 0-No)	② Are we accessing text (0) or data (1) from memory?	③ Are we using memory? (1-Y/0-N)	④ Are we writing to memory? (1-Y/0-N)	⑤ Are we storing a new instr? (1-Y/0-N)	⑥ Are we writing to reg B (0) or the dest reg (1)?	⑦ Are we writing data (0) or ALU result (1) to register?	⑧ Are we writing to reg file? (1-Y/0-N)	⑨ Are we operating on PC (0) or [reg A] (1)?	⑩, ⑪ Are we operating on [reg B] (00), 1 (01), 0 (10), or the offset (11)?	⑫ Are we adding (0) or NORing (1)?	Next state? ST3	ST2	ST1	ST0	CURR STATE		Store instr. in IR	Store PC+1 in ALUR
0	0	1 (get instr)	0	1	X	X	0	X	01 (PC+1 in ALU)	0	0	0	0	1	0	Fetch instr.	FETCH		
1	X	0	X	0	X	X	0	X	X	X	X	0	0	1	1	Read instr.	DECODE	PC ← PC+1; branch to state; read RA, RB	
0	X	0	X	0	X	X	0	1	00	0	X	0	1	1	2	ADD 3	EXEC	[reg A] + [reg B] in ALUR	
0	X	0	X	0	1	1	1	X	X	X	0	0	0	0	3	ADD 4	WRITEBACK	ALUR to dest Reg	
0	X	0	X	0	X	X	0	1	00	1	X	0	1	1	4	NOR 3	EXEC	[reg A] NOR [reg B] in ALUR	
0	X	0	X	0	1	1	1	X	X	X	0	0	0	0	5	NOR 4	WRITEBACK	ALUR to dest reg	
0	X	0	X	0	X	X	0	1	11	0	X	0	1	1	6	LW 3	EXEC	[reg A] + offset in ALUR	
0	1	1	0	0	X	X	0	X	X	X	X	1	0	0	7	LW 4	MEM ACCESS	access data [ALUR]	
0	X	0	X	0	0	0	1	X	X	X	X	0	0	0	8	LW 5	WRITEBACK	write data to [reg B]	
0	X	0	X	0	X	X	0	1	11	0	X	1	0	0	1	9	SW 3	EXEC	[reg A] + offset in ALUR	
0	1	1	1	0	X	X	0	X	X	X	X	0	0	0	0	10	SW 4	MEM ACCESS	write [reg B] to data [ALUR]	
0	X	0	X	0	X	X	0	0	11	0	X	1	0	1	0	11	BEQ 3	EXEC	[PC+1] (alr in PC) + offset in ALUR	
0 (use EQ)	X	0	X	0	X	X	0	1	00	X	X	0	0	0	0	12	BEQ 4	WRITEBACK	[reg A], [reg B] to ALU for EQ	
PC en		MUX addr		Mem en		Mem R/W		IR en	MUX dest		MUX rdata	Reg en		MUX ALUS	MUX ALUZ		ALU op		

Logic Gate Building Blocks

- INV using NOR
- INV using NAND
- AND using NOR
- AND using NAND
- OR using NOR
- 2 x 1 MUX using NOR
- OR using NAND