



北京大学
PEKING UNIVERSITY

22530007

人工智能与芯片设计

6- Design Flow (一款数字处理器芯片的诞生)

燕博南
2023秋

Outline

- Design Partitioning
- MIPS Processor Example
 - Architecture
 - Microarchitecture
 - Logic Design
 - Circuit Design
 - Physical Design
- Fabrication, Packaging, Testing

Coping with Complexity

- How to design System-on-Chip?
 - Many millions (even billions!) of transistors
 - Tens to hundreds of engineers
- Structured Design
- Design Partitioning

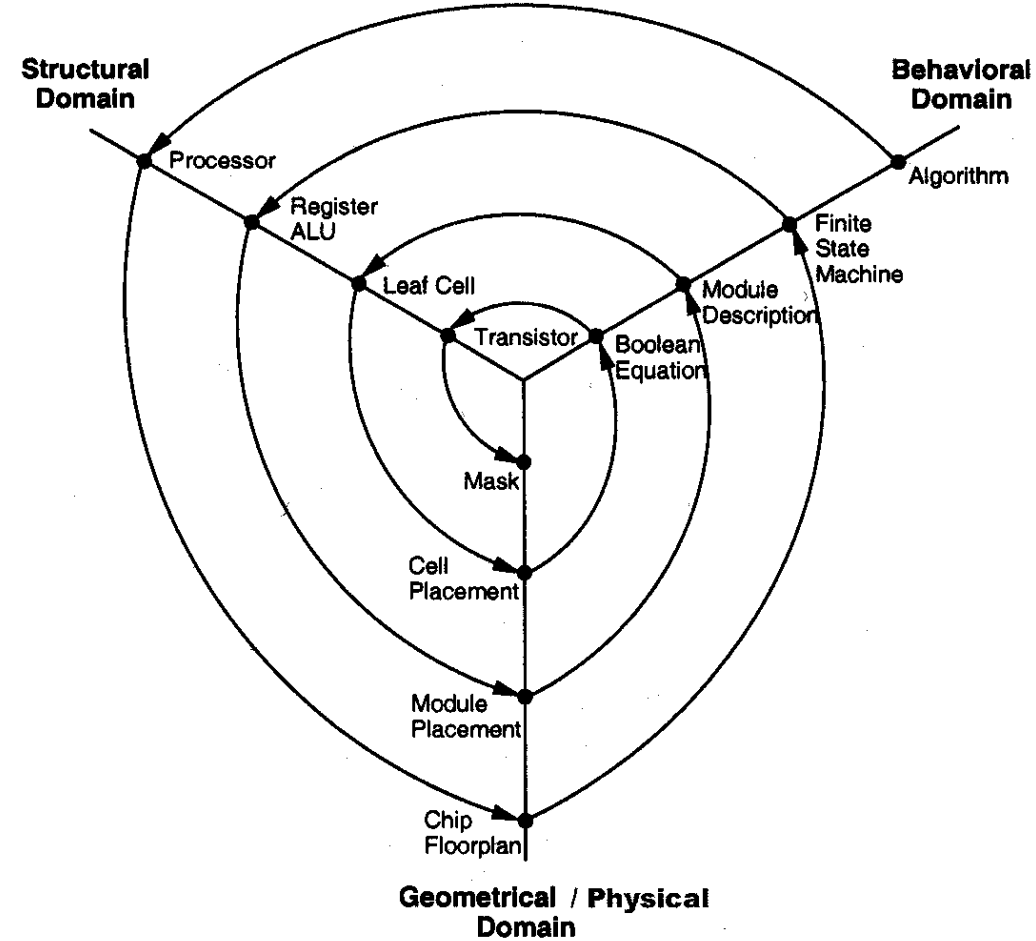
Structured Design

- **Hierarchy: Divide and Conquer**
 - Recursively system into modules
- **Regularity**
 - Reuse modules wherever possible
 - Ex: Standard cell library
- **Modularity: well-formed interfaces**
 - Allows modules to be treated as black boxes
- **Locality**
 - Physical and temporal

Design Partitioning

- **Architecture:** User's perspective, what does it do?
 - Instruction set, registers
 - MIPS, x86, Alpha, PIC, ARM, ...
- **Microarchitecture**
 - Single cycle, multicycle, pipelined, superscalar?
- **Logic:** how are functional blocks constructed
 - Ripple carry, carry lookahead, carry select adders
- **Circuit:** how are transistors used
 - Complementary CMOS, pass transistors, domino
- **Physical:** chip layout
 - Datapaths, memories, random logic

Gajski Y-Chart



MIPS Architecture

- Microprocessor without Interlocked Pipeline Stages
- MIPS is a 32-bit architecture with 32 registers
 - Consider 8-bit subset using 8-bit datapath
 - Only implement 8 registers (\$0 - \$7)
 - \$0 hardwired to 00000000
 - 8-bit program counter
- Predecessor of RISC-V

Instruction Set

Table 1.7 MIPS instruction set (subset supported)

Instruction	Function	Encoding	op	funct
add \$1, \$2, \$3	addition: $\$1 \rightarrow \$2 + \$3$	R	000000	100000
sub \$1, \$2, \$3	subtraction: $\$1 \rightarrow \$2 - \$3$	R	000000	100010
and \$1, \$2, \$3	bitwise and: $\$1 \rightarrow \$2 \text{ and } \$3$	R	000000	100100
or \$1, \$2, \$3	bitwise or: $\$1 \rightarrow \$2 \text{ or } \$3$	R	000000	100101
slt \$1, \$2, \$3	set less than: $\$1 \rightarrow 1 \text{ if } \$2 < \$3$ $\$1 \rightarrow 0 \text{ otherwise}$	R	000000	101010
addi \$1, \$2, imm	add immediate: $\$1 \rightarrow \$2 + \text{imm}$	I	001000	n/a
beq \$1, \$2, imm	branch if equal: $\text{PC} \rightarrow \text{PC} + \text{imm}^a$	I	000100	n/a
j destination	jump: PC_destination^a	J	000010	n/a
lb \$1, imm(\$2)	load byte: $\$1 \rightarrow \text{mem}[\$2 + \text{imm}]$	I	100000	n/a
sb \$1, imm(\$2)	store byte: $\text{mem}[\$2 + \text{imm}] \rightarrow \1	I	110000	n/a

Instruction Encoding

- 32-bit instruction encoding
 - Requires four cycles to fetch on 8-bit datapath

format	example	encoding					
R	add \$rd, \$ra, \$rb	6	5	5	5	5	6
		0	ra	rb	rd	0	funct
I	beq \$ra, \$rb, imm	6	5	5	16		
		op	ra	rb	imm		
J	j dest	6	26				
		op	dest				

Fibonacci (C)

$$f_0 = 1; f_{-1} = -1$$

$$f_n = f_{n-1} + f_{n-2}$$

$$f = 1, 1, 2, 3, 5, 8, 13, \dots$$

```
int fib(void)
{
    int n = 8;           /* compute nth Fibonacci number */
    int f1 = 1, f2 = -1; /* last two Fibonacci numbers */

    while (n != 0) {      /* count down to n = 0 */
        f1 = f1 + f2;
        f2 = f1 - f2;
        n = n - 1;
    }
    return f1;
}
```

Fibonacci (Assembly)

- 1st statement: $n = 8$
- How do we translate this to assembly?

```
# fib.asm
# Register usage: $3: n $4: f1 $5: f2
# return value written to address 255
fib:  addi $3, $0, 8      # initialize n=8
      addi $4, $0, 1      # initialize f1 = 1
      addi $5, $0, -1     # initialize f2 = -1
loop: beq $3, $0, end     # Done with loop if n = 0
      add $4, $4, $5      # f1 = f1 + f2
      sub $5, $4, $5      # f2 = f1 - f2
      addi $3, $3, -1     # n = n - 1
      j loop              # repeat until done
end:  sb $4, 255($0)      # store result in address 255
```

Fibonacci (Binary)

- 1st statement: addi \$3, \$0, 8
- How do we translate this to machine language?
 - Hint: use instruction encodings below

format	example	encoding					
		6	5	5	5	5	6
R	add \$rd, \$ra, \$rb	0	ra	rb	rd	0	funct
		6	5	5	16		
I	beq \$ra, \$rb, imm	op	ra	rb	imm		
		6	26				
J	j dest	op	dest				

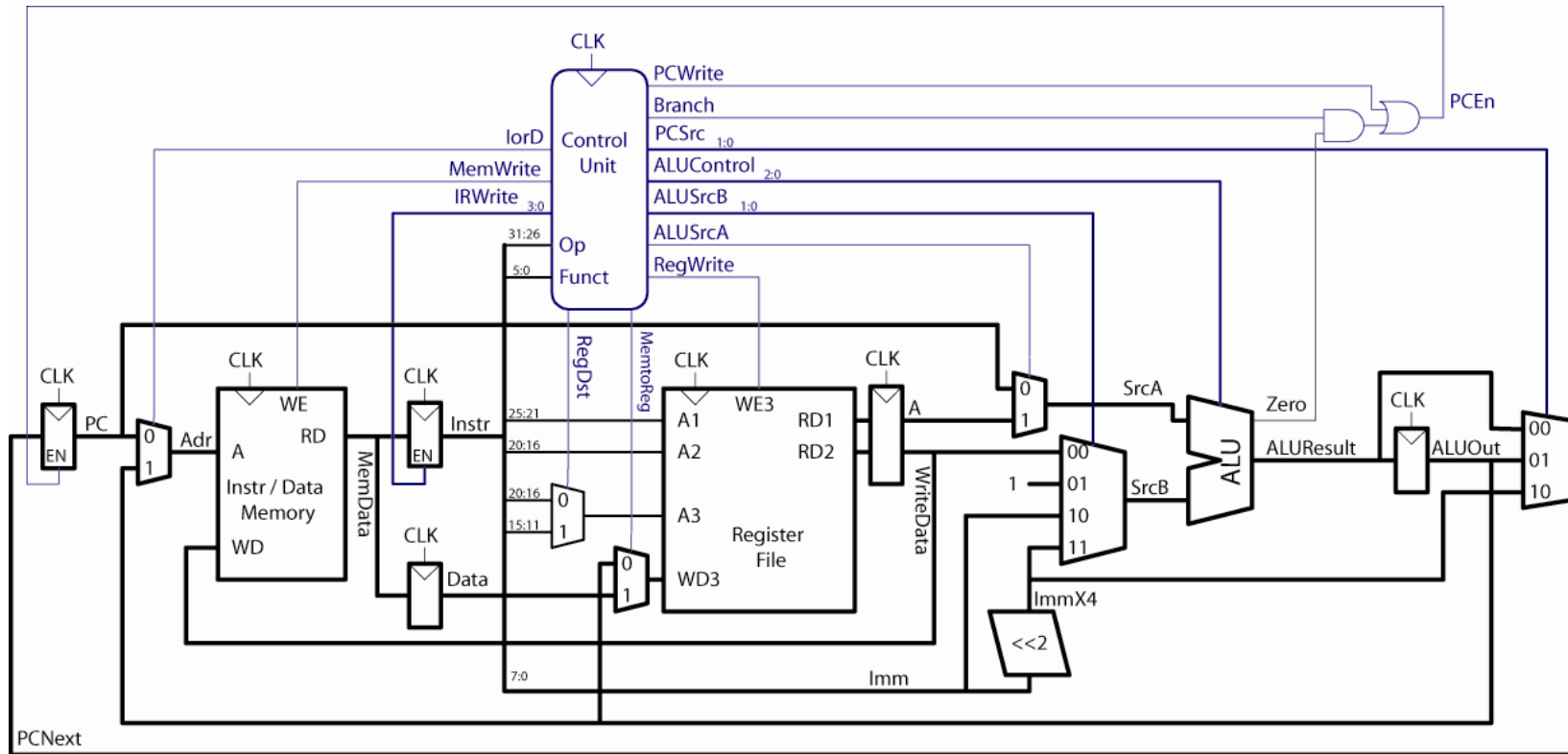
Fibonacci (Binary)

- Machine language program

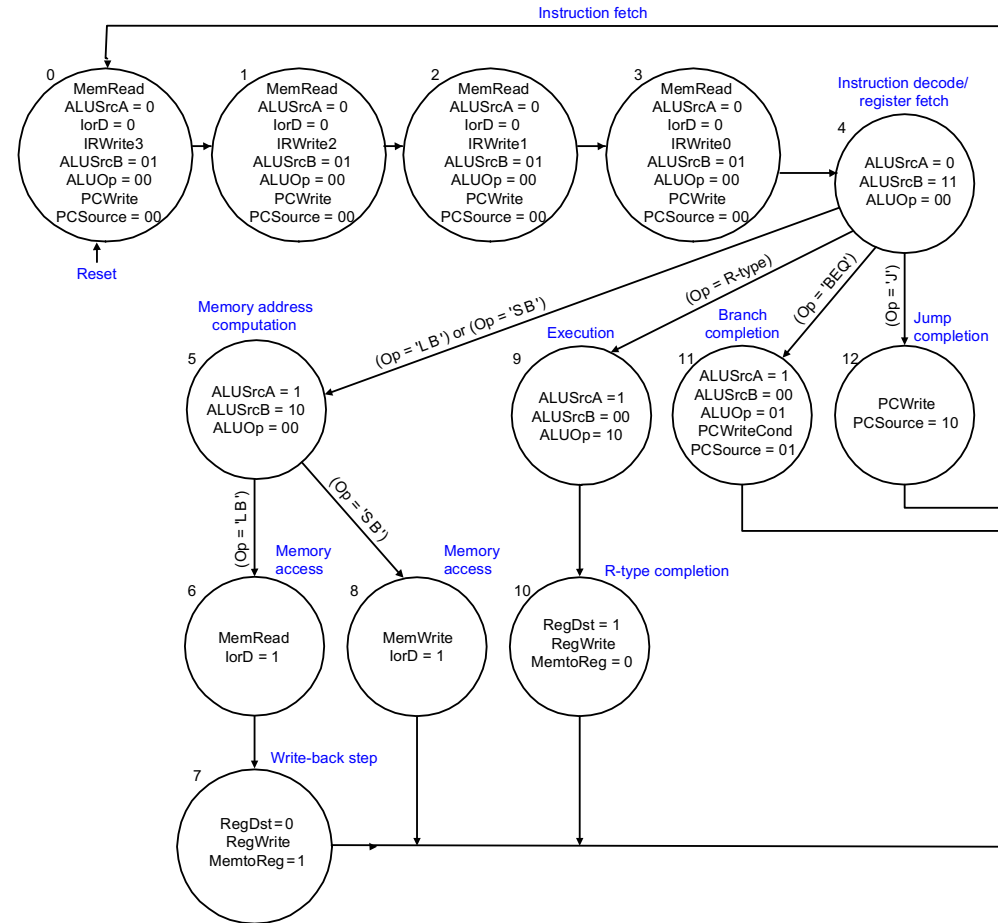
Instruction	Binary Encoding			Hexadecimal Encoding
addi \$3, \$0, 8	001000	00000	00011 00000000000001000	20030008
addi \$4, \$0, 1	001000	00000	00100 00000000000000001	20040001
addi \$5, \$0, -1	001000	00000	00101 11111111111111111	2005ffff
beq \$3, \$0, end	000100	00011	00000 00000000000000101	10600005
add \$4, \$4, \$5	000000	00100	00101 00100 00000 100000	00852020
sub \$5, \$4, \$5	000000	00100	00101 00101 00000 100010	00852822
addi \$3, \$3, -1	001000	00011	00011 11111111111111111	2063ffff
j loop	000010	00000000000000000000000000000011		08000003
sb \$4, 255(\$0)	110000	00000	00100 00000000111111111	a00400ff

MIPS Microarchitecture

- Multicycle μ architecture ([Paterson04], [Harris07])

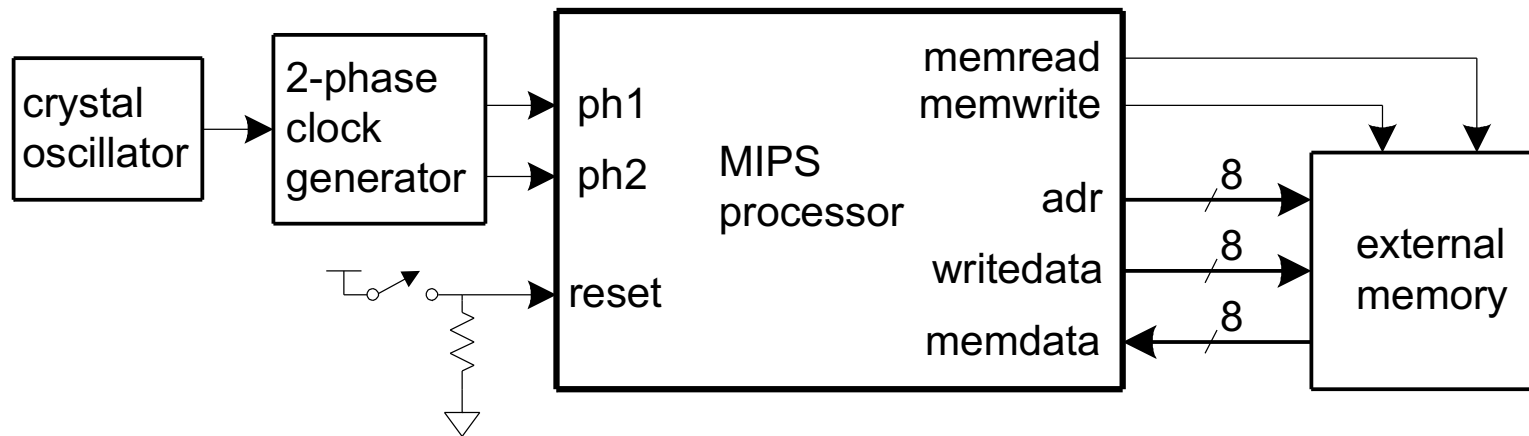


Multicycle Controller

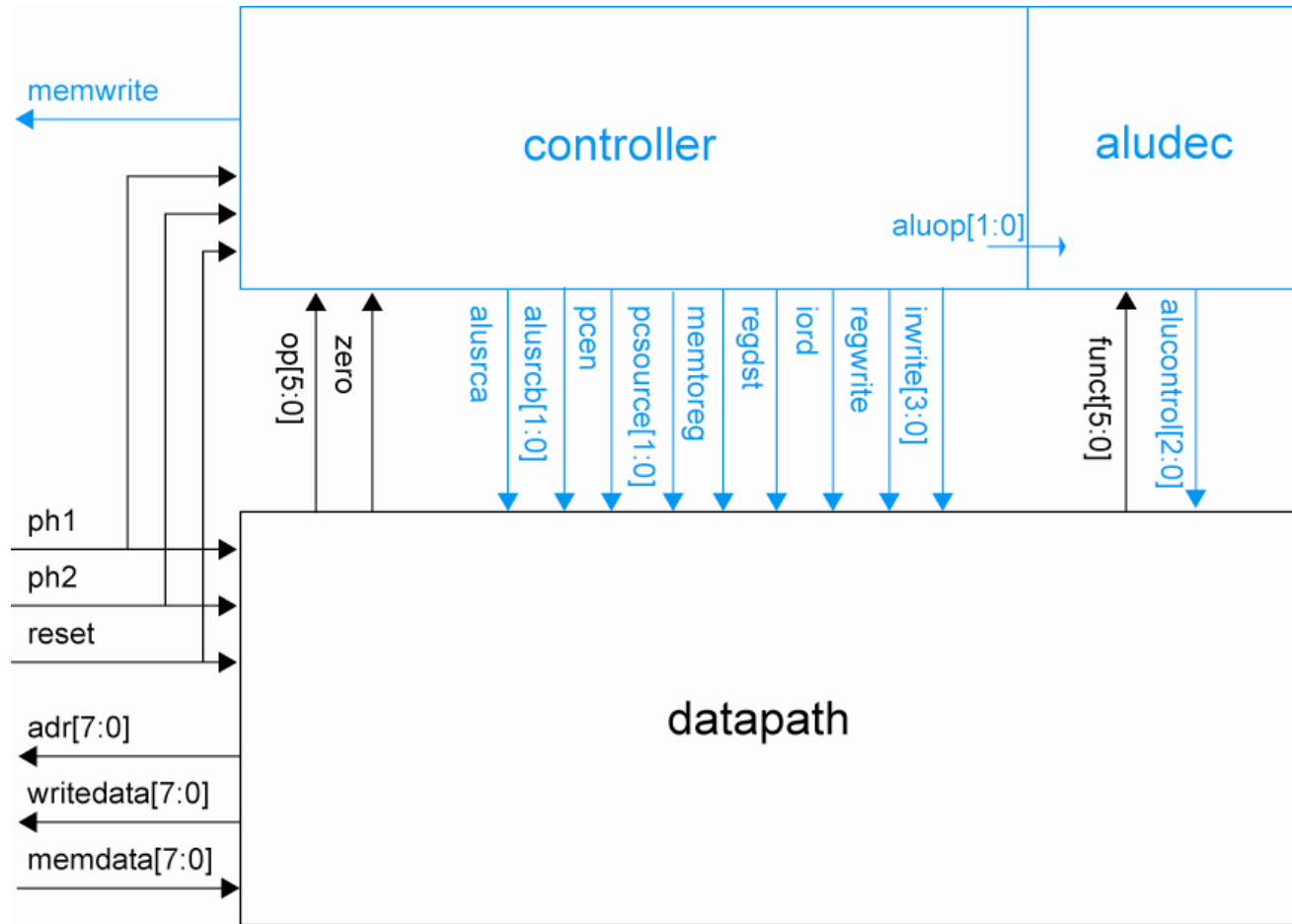


Logic Design

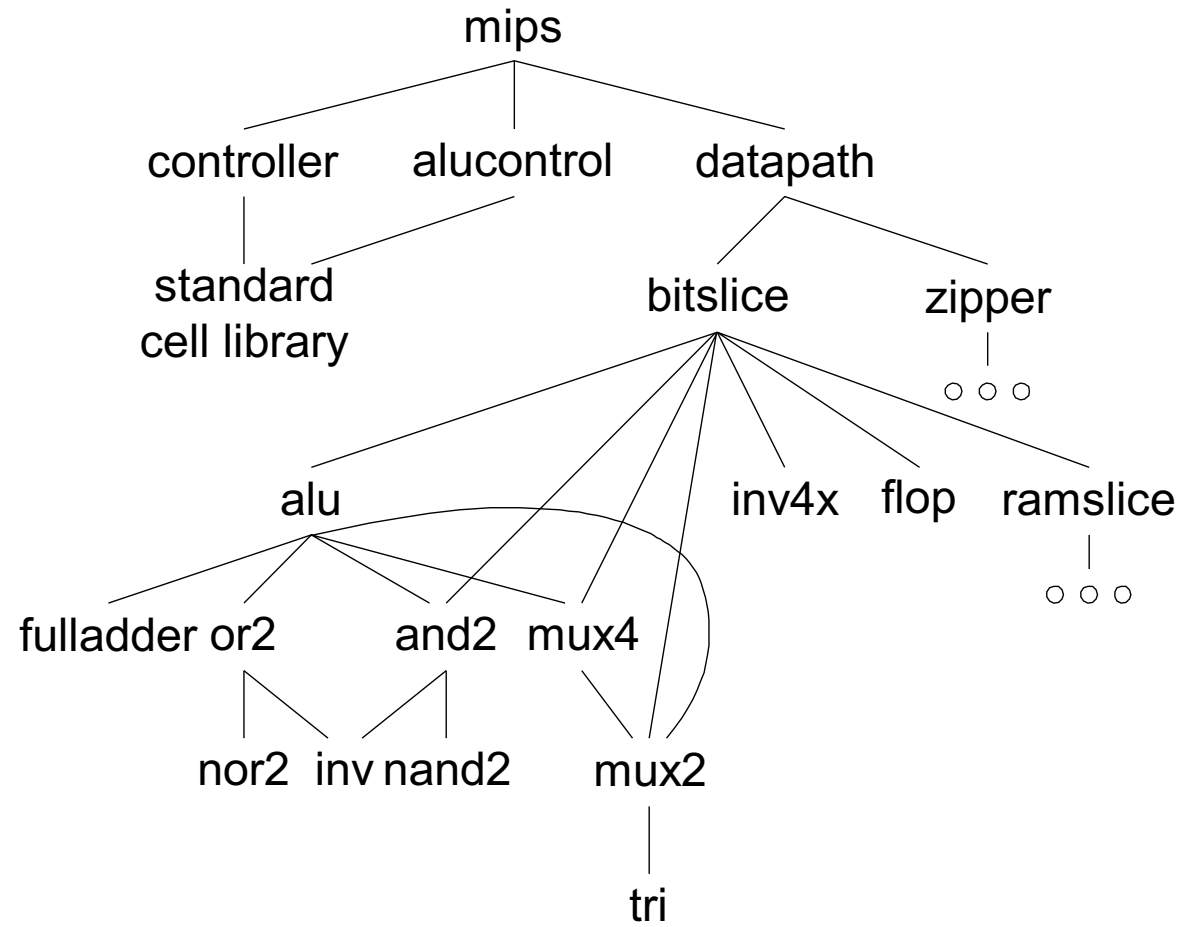
- Start at top level
 - Hierarchically decompose MIPS into units
- Top-level interface



Block Diagram



Hierarchical Design



HDLs

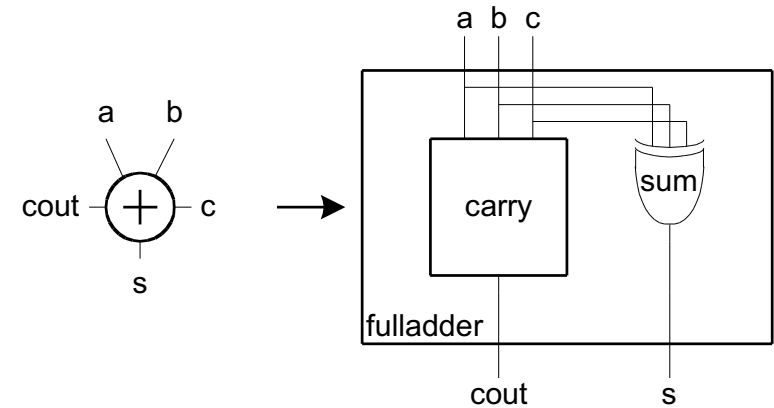
- Hardware Description Languages
 - Widely used in logic design
 - Verilog, VHDL, Chisel HGL
- Describe hardware using code
 - Document logic functions
 - Simulate logic before building
 - Synthesize code into gates and layout
 - Requires a library of standard cells

Verilog Example

```
module fulladder(input  a, b, c,  
                  output s, cout);
```

```
    sum      s1(a, b, c, s);  
    carry    c1(a, b, c, cout);  
endmodule
```

```
module carry(input  a, b, c,  
            output cout)  
  
    assign cout = (a&b) | (a&c) | (b&c);  
endmodule
```

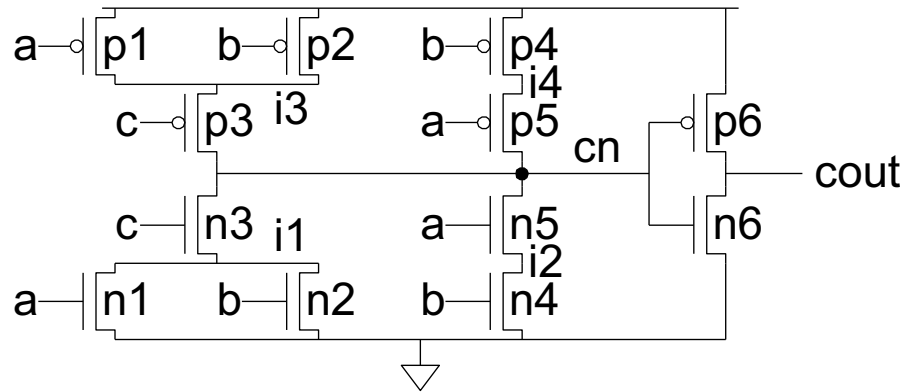
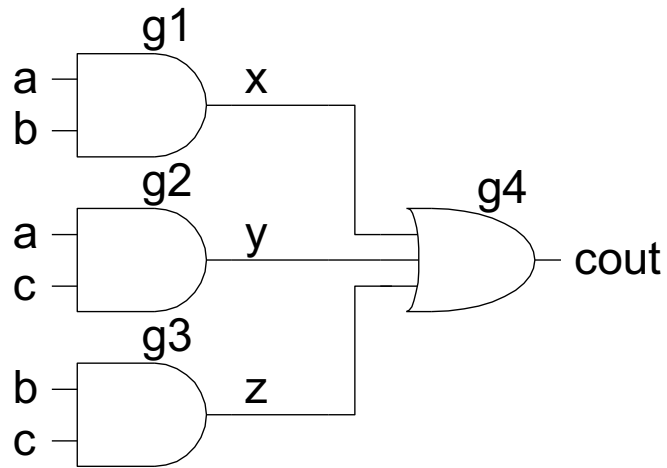


Circuit Design

- How should logic be implemented?
 - NANDs and NORs vs. ANDs and ORs?
 - Fan-in and fan-out?
 - How wide should transistors be?
- These choices affect speed, area, power
- Logic synthesis makes these choices for you
 - Good enough for many applications
 - Hand-crafted circuits are still better

Example: Carry Logic

- **assign** cout = (a&b) | (a&c) | (b&c);



Transistors? Gate Delays?

Gate-level Netlist

```
module carry(input  a, b, c,  
             output cout)
```

```
    wire      x, y, z;
```

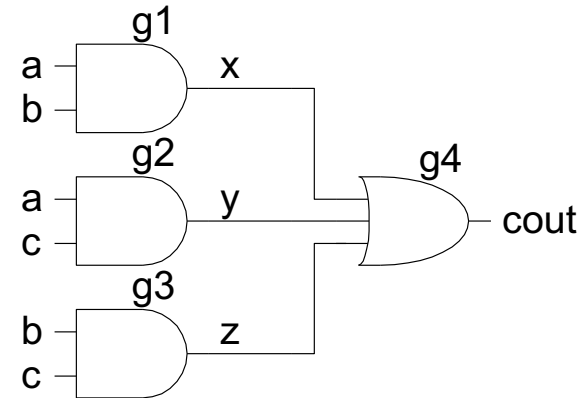
```
    and g1(x, a, b);
```

```
    and g2(y, a, c);
```

```
    and g3(z, b, c);
```

```
    or  g4(cout, x, y, z);
```

```
endmodule
```

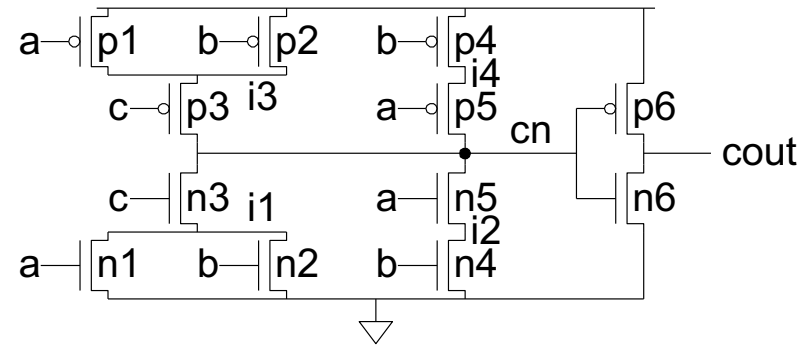


Transistor-Level Netlist

```
module carry(input  a, b, c,
              output cout)

    wire      i1, i2, i3, i4, cn;

    tranif1 n1(i1, 0, a);
    tranif1 n2(i1, 0, b);
    tranif1 n3(cn, i1, c);
    tranif1 n4(i2, 0, b);
    tranif1 n5(cn, i2, a);
    tranif0 p1(i3, 1, a);
    tranif0 p2(i3, 1, b);
    tranif0 p3(cn, i3, c);
    tranif0 p4(i4, 1, b);
    tranif0 p5(cn, i4, a);
    tranif1 n6(cout, 0, cn);
    tranif0 p6(cout, 1, cn);
endmodule
```



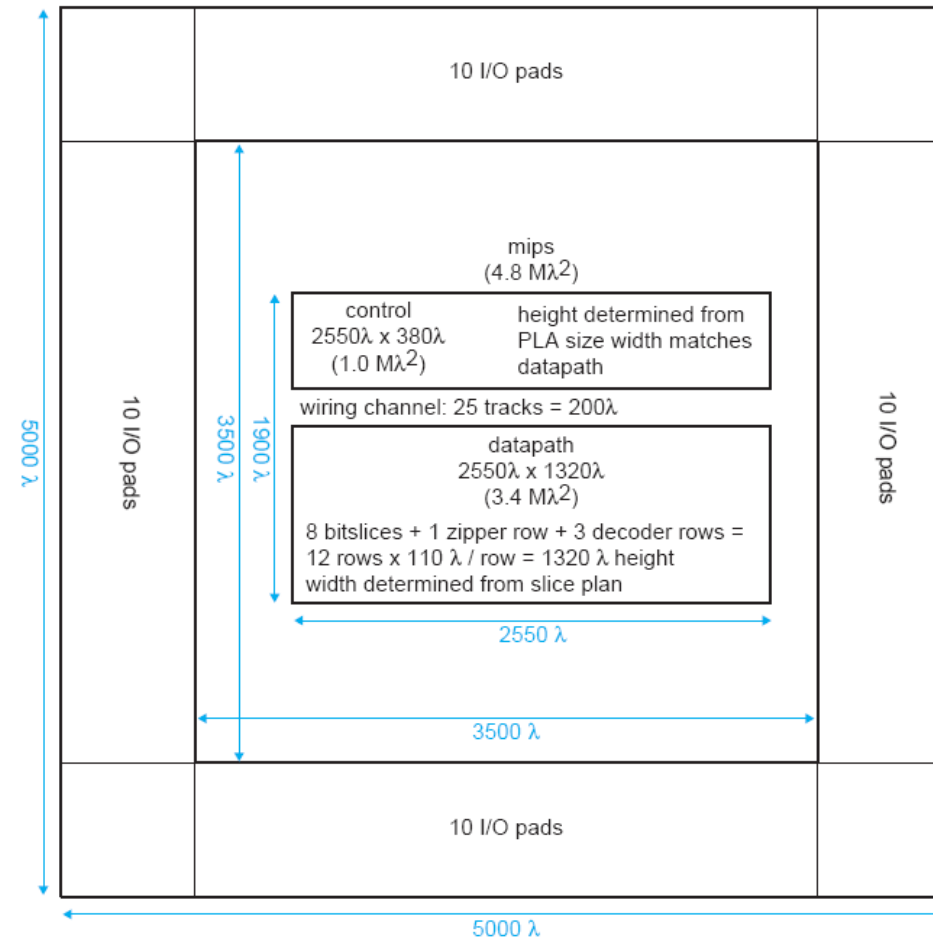
SPICE Netlist

```
.SUBCKT CARRY A B C COUT VDD GND
MN1 I1 A GND GND NMOS W=1U L=0.18U AD=0.3P AS=0.5P
MN2 I1 B GND GND NMOS W=1U L=0.18U AD=0.3P AS=0.5P
MN3 CN C I1 GND NMOS W=1U L=0.18U AD=0.5P AS=0.5P
MN4 I2 B GND GND NMOS W=1U L=0.18U AD=0.15P AS=0.5P
MN5 CN A I2 GND NMOS W=1U L=0.18U AD=0.5P AS=0.15P
MP1 I3 A VDD VDD PMOS W=2U L=0.18U AD=0.6P AS=1 P
MP2 I3 B VDD VDD PMOS W=2U L=0.18U AD=0.6P AS=1P
MP3 CN C I3 VDD PMOS W=2U L=0.18U AD=1P AS=1P
MP4 I4 B VDD VDD PMOS W=2U L=0.18U AD=0.3P AS=1P
MP5 CN A I4 VDD PMOS W=2U L=0.18U AD=1P AS=0.3P
MN6 COUT CN GND GND NMOS W=2U L=0.18U AD=1P AS=1P
MP6 COUT CN VDD VDD PMOS W=4U L=0.18U AD=2P AS=2P
CI1 I1 GND 2FF
CI3 I3 GND 3FF
CA A GND 4FF
CB B GND 4FF
CC C GND 2FF
CCN CN GND 4FF
CCOUT COUT GND 2FF
.ENDS
```

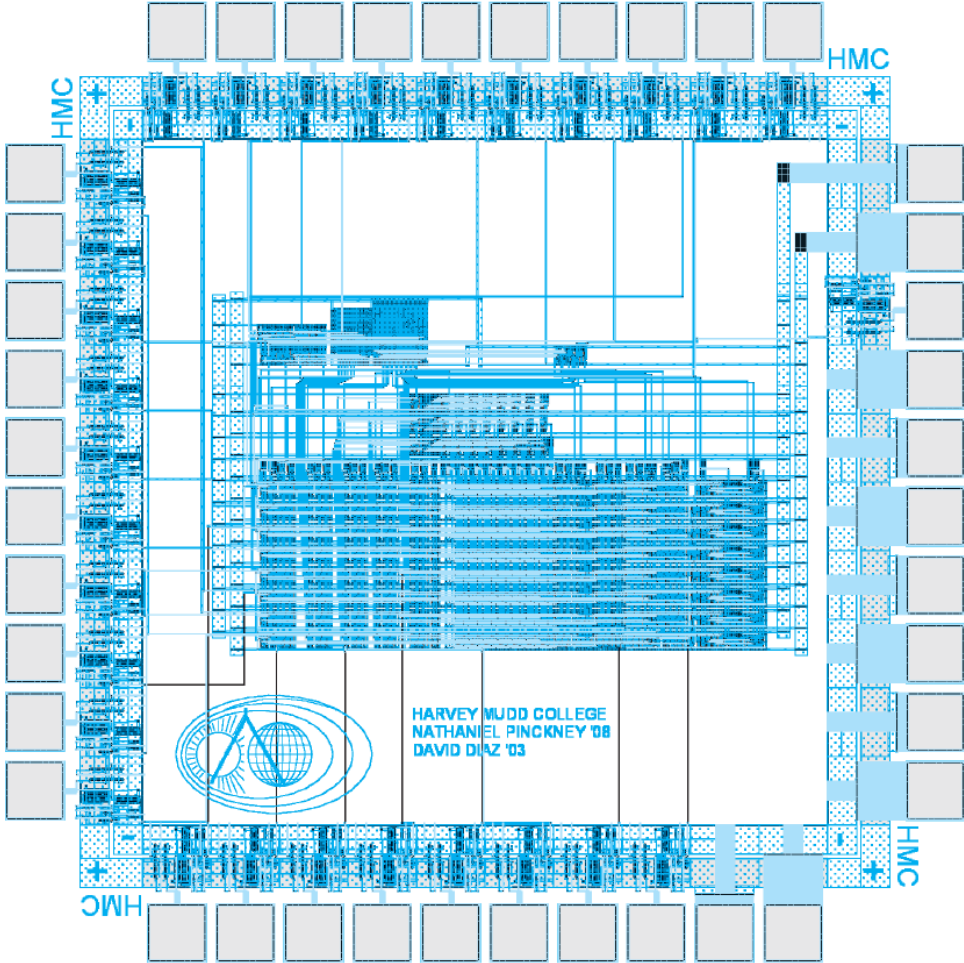
Physical Design

- Floorplan
- Standard cells
 - Place & route
- Datapaths
 - Slice planning
- Area estimation

MIPS Floorplan

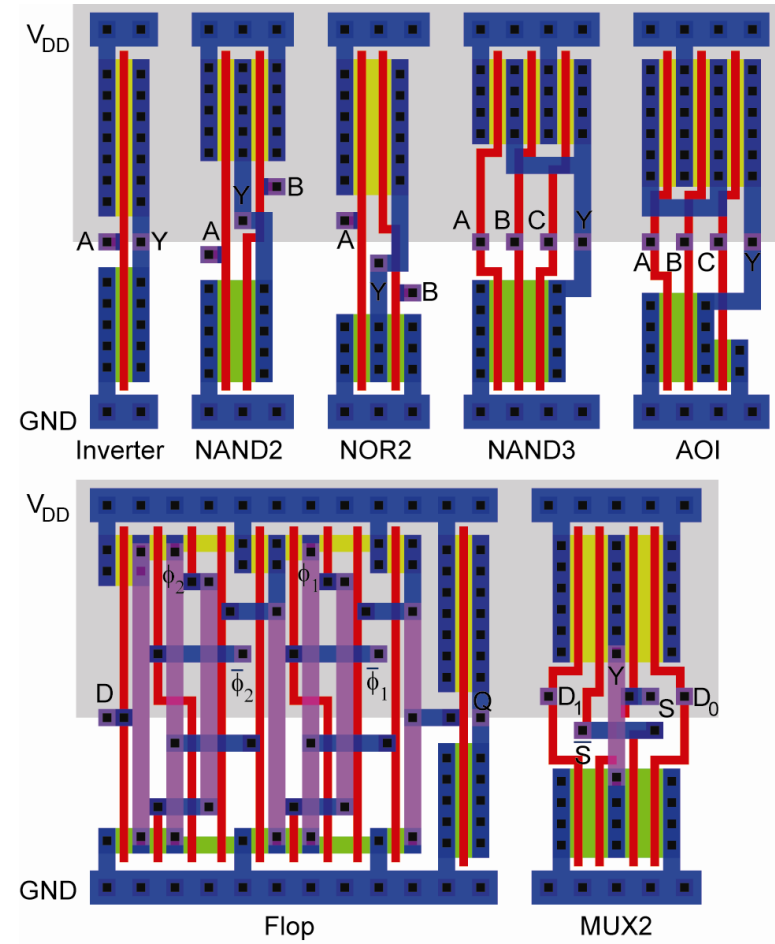


MIPS Layout



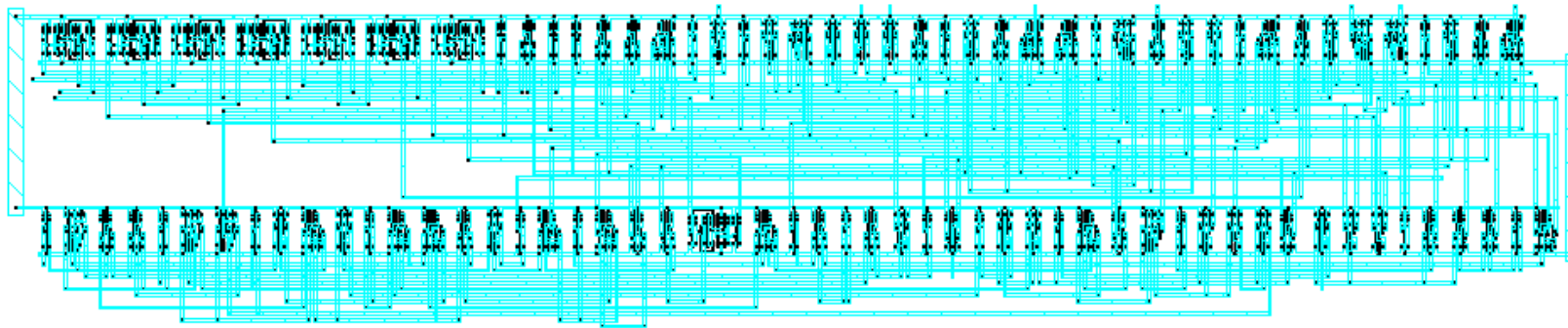
Standard Cells

- Uniform cell height
- Uniform well height
- M1 V_{DD} and GND rails
- M2 Access to I/Os
- Well / substrate taps
- Exploits regularity



Synthesized Controller

- Synthesize HDL into gate-level netlist
- Place & Route using standard cell library



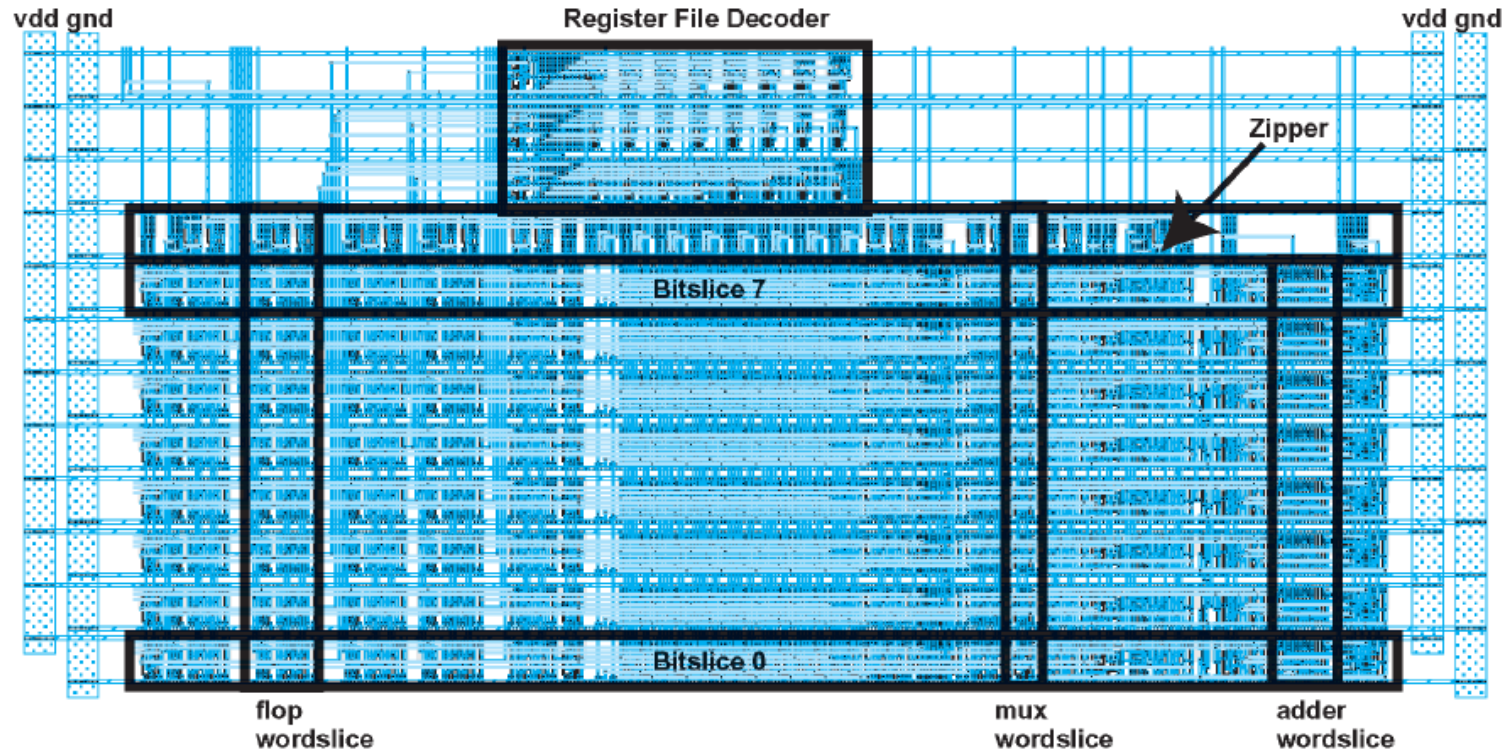
Pitch Matching

- Synthesized controller area is mostly wires
 - Design is smaller if wires run through/over cells
 - Smaller = faster, lower power as well!
- Design snap-together cells for datapaths and arrays
 - Plan wires into cells
 - Connect by abutment
 - Exploits locality
 - Takes lots of effort

A	A	A	A	B
A	A	A	A	B
A	A	A	A	B
A	A	A	A	B
C		C		D

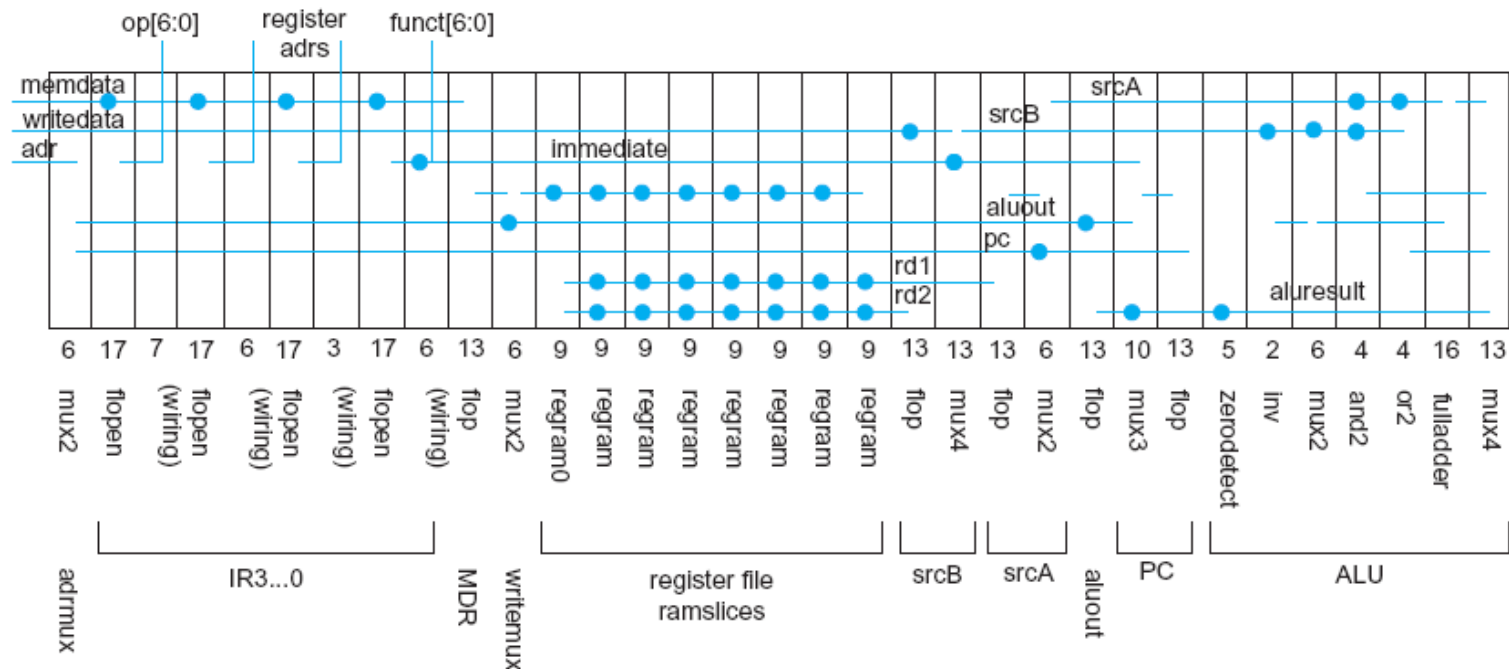
MIPS Datapath

- 8-bit datapath built from 8 bitslices (regularity)
- Zipper at top drives control signals to datapath



Slice Plans

- Slice plan for bitslice
 - Cell ordering, dimensions, wiring tracks
 - Arrange cells for wiring locality



Area Estimation

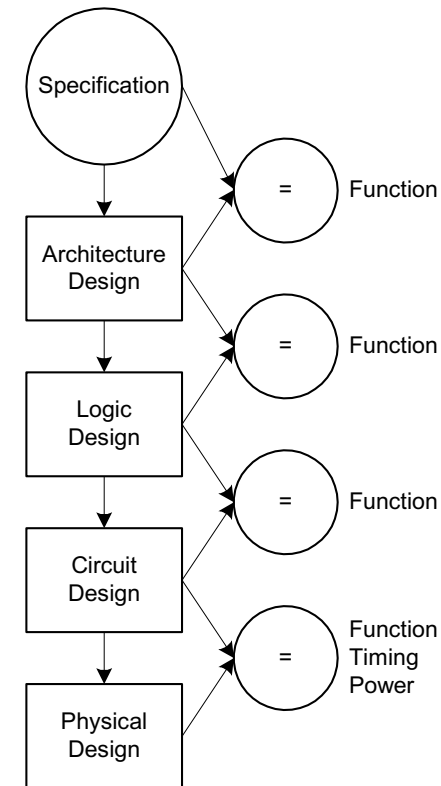
- Need area estimates to make floorplan
 - Compare to another block you already designed
 - Or estimate from transistor counts
 - Budget room for large wiring tracks
 - Your mileage may vary; derate by 2x for class.

Table 1.10 Typical layout densities

Element	Area
random logic (2-level metal process)	1000 – 1500 λ^2 / transistor
datapath	250 – 750 λ^2 / transistor or 6 WL + 360 λ^2 / transistor
SRAM	1000 λ^2 / bit
DRAM (in a DRAM process)	100 λ^2 / bit
ROM	100 λ^2 / bit

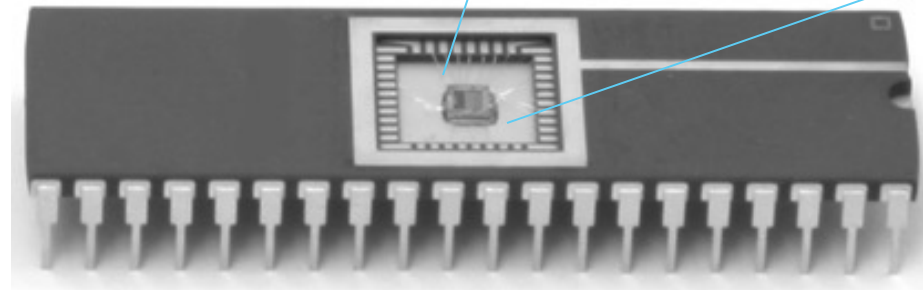
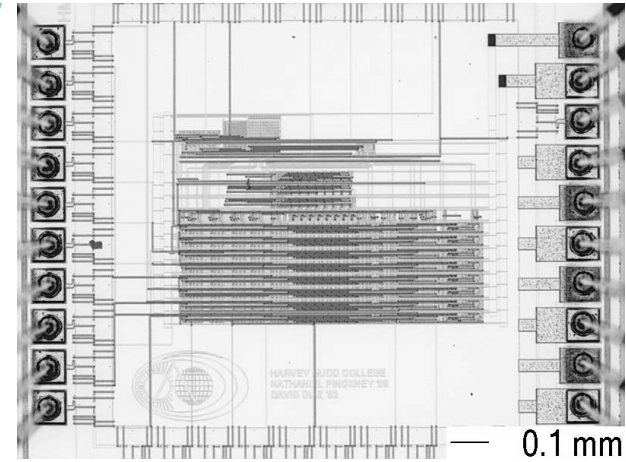
Design Verification

- Fabrication is slow & expensive
 - MOSIS 0.6 μ m: \$1000, 3 months
 - 65 nm: \$3M, 1 month
- Debugging chips is very hard
 - Limited visibility into operation
- Prove design is right before building!
 - Logic simulation
 - Ckt. simulation / formal verification
 - Layout vs. schematic comparison
 - Design & electrical rule checks
- Verification is > 50% of effort on most chips!



Fabrication & Packaging

- Tapeout final layout
- Fabrication
 - 6, 8, 12" wafers
 - Optimized for throughput, not latency (10 weeks!)
 - Cut into individual dice
- Packaging
 - Bond gold wires from die I/O pads to package

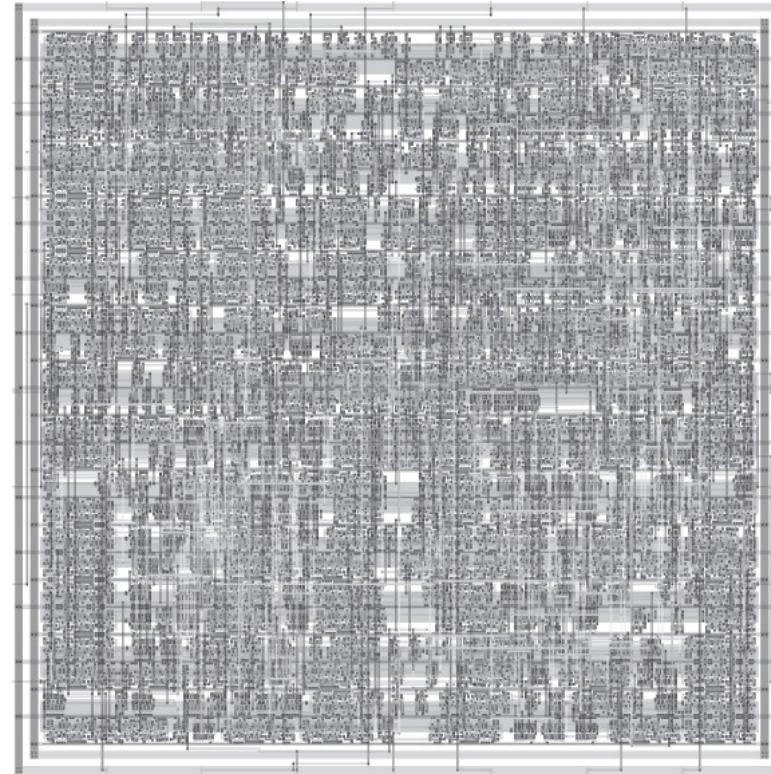
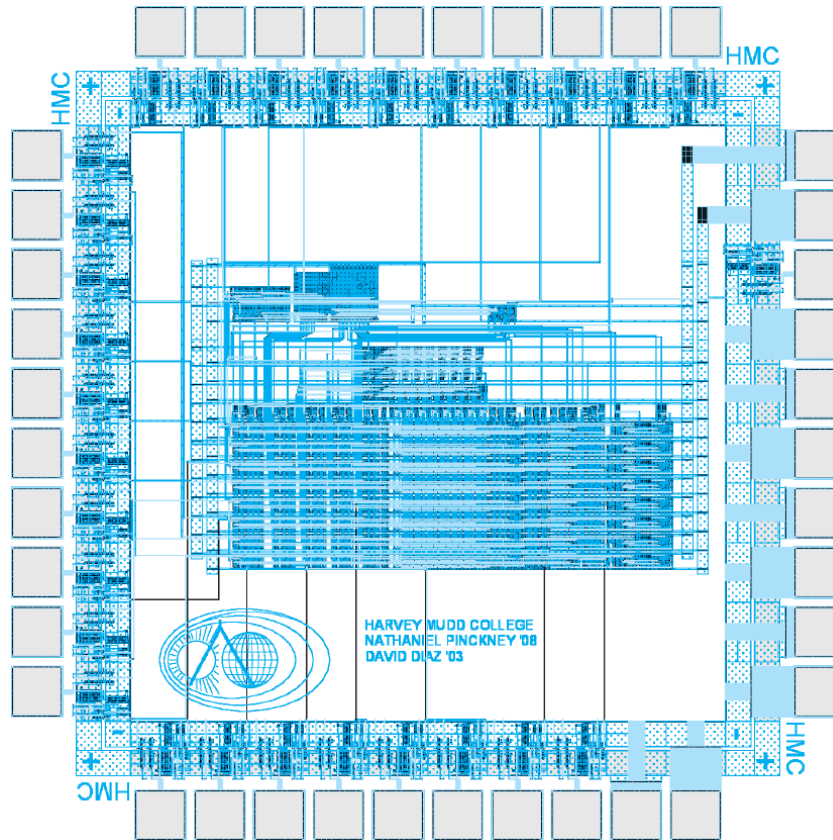


Testing

- Test that chip operates
 - Design errors
 - Manufacturing errors
- A single dust particle or wafer defect kills a die
 - Yields from 90% to $< 10\%$
 - Depends on die size, maturity of process
 - Test each part before shipping to customer

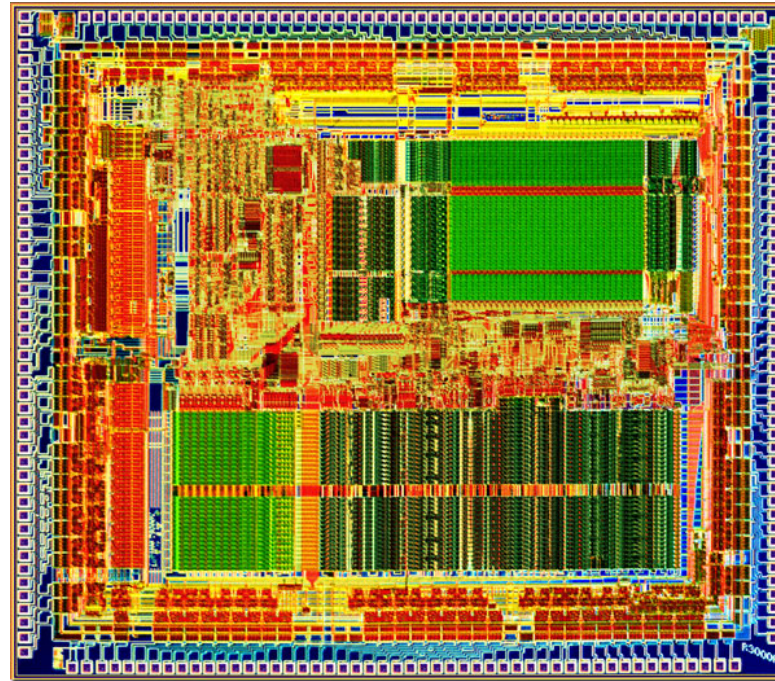
Custom vs. Synthesis

❑ 8-bit Implementations



MIPS R3000 Processor

- 32-bit 2nd generation commercial processor (1988)
- Led by John Hennessy (Stanford, MIPS Founder)
- 32-64 KB Caches
- 1.2 μm process
- 111K Transistors
- Up to 12-40 MHz
- 66 mm^2 die
- 145 I/O Pins
- $V_{\text{DD}} = 5 \text{ V}$
- 4 Watts
- SGI Workstations



http://gecko54000.free.fr/?documentations=1988_MIPS_R3000