Combinational Logic Design Process

1. **Understand the Problem**
   - what is the circuit supposed to do?
   - write down inputs (data, control) and outputs
   - draw block diagram or other picture

2. **Formulate the Problem in terms of a truth table or other suitable design representation**
   - truth table, Boolean algebra, etc.

3. **Choose Implementation Target**
   - PAL, PLA, Mux, Decoder, Discrete Gates

4. **Follow Implementation Procedure**
   - K-maps, Boolean algebra

Process Line Control Example

Statement of the Problem

Rods of varying length (+/-10%) travel on conveyor belt
Mechanical arm pushes rods within spec (+/-5%) to one side
Second arm pushes rods too long to other side
Rods too short stay on belt

3 light barriers (light source + photocell) as sensors

Design combinational logic to activate the arms

Understanding the Problem

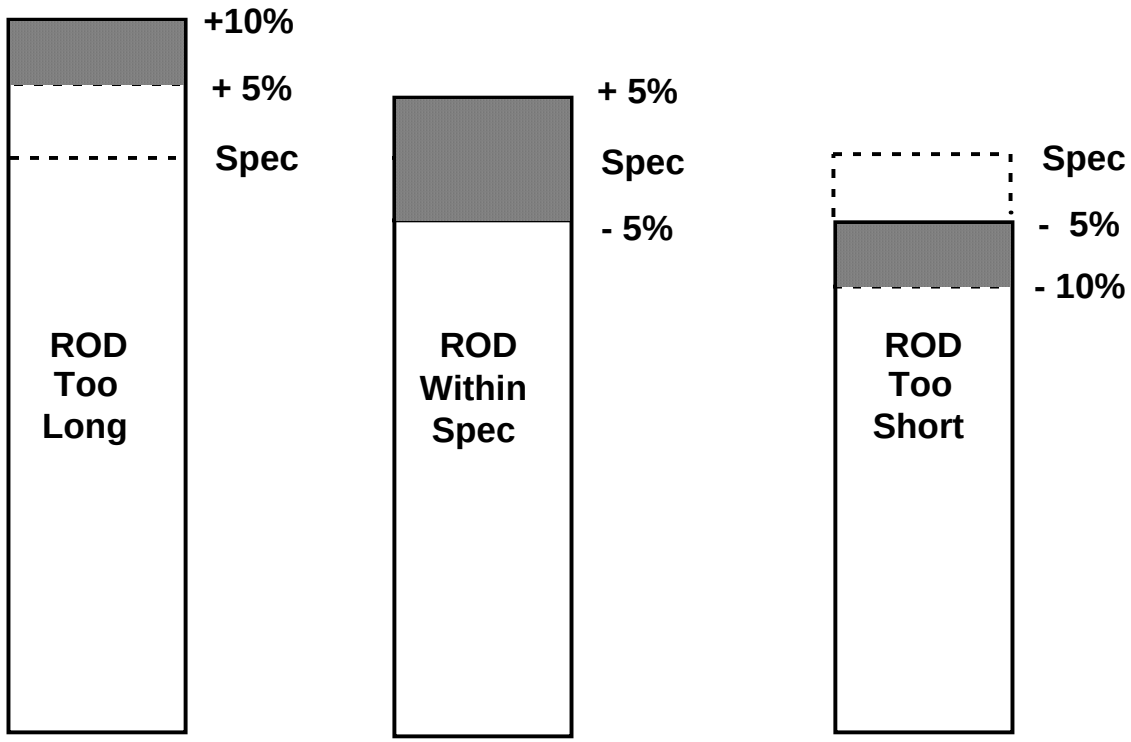
Inputs are three sensors, outputs are two arm control signals

Assume sensor reads "1" when tripped, "0" otherwise

Call sensors A, B, C

Draw a picture!

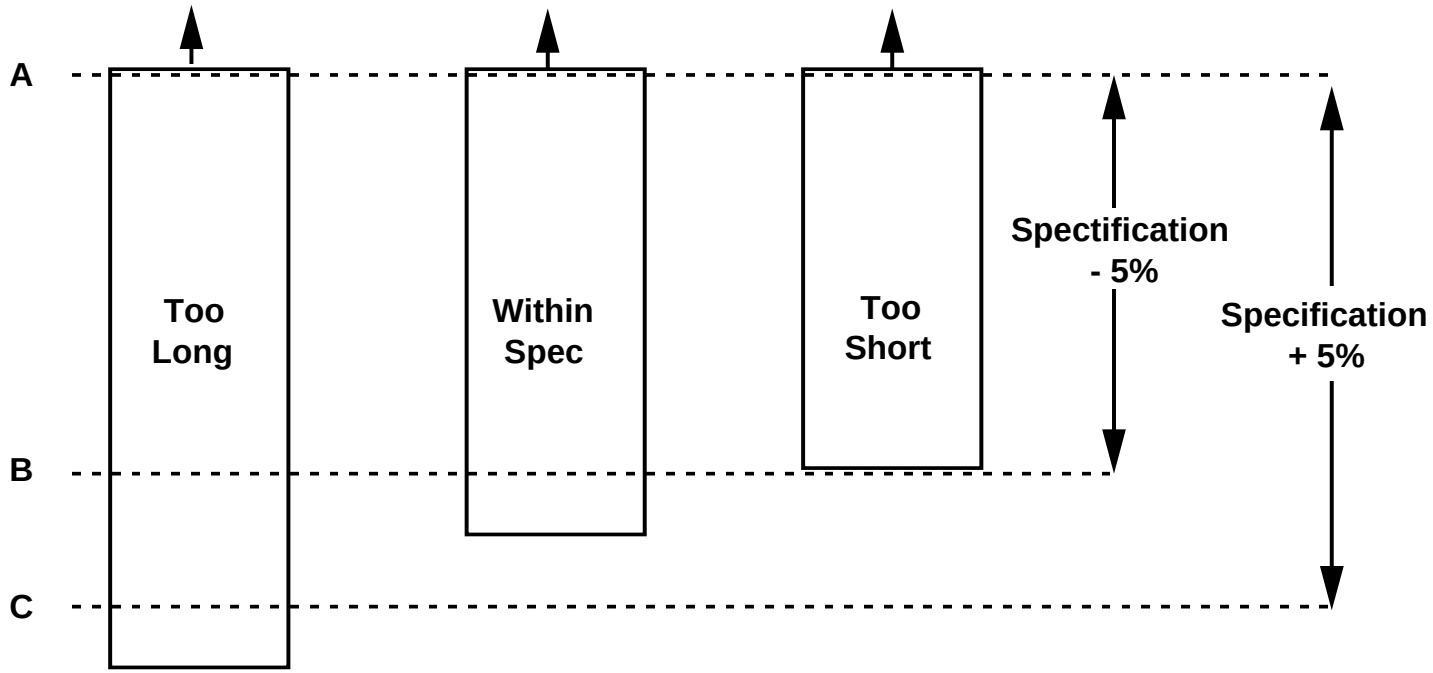
Process Line Control Example (cont.)



Where to place the light sensors A, B, and C to distinguish among the three cases?

Assume that A detects the leading edge of the rod on the conveyor

Process Line Control Example (cont.)



A to B distance place apart at specification - 5%

A to C distance placed apart at specification +5%

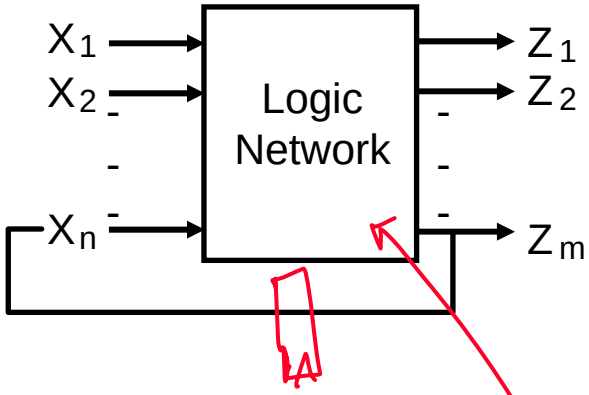
Process Line Control Example (cont.)

A	B	C	Meaning	Accept	Long
0	0	0	No bar	0/x	0/X
0	0	1	Rod arriving	0	0
0	1	0	?	0	0
0	1	1	Arriving	0	0
1	0	0	Too short	0	0
1	0	1	?	0	0
1	1	0	just right	1	0
1	1	1	long	0	1

Accept = $AB\bar{C}$

Long = ABC

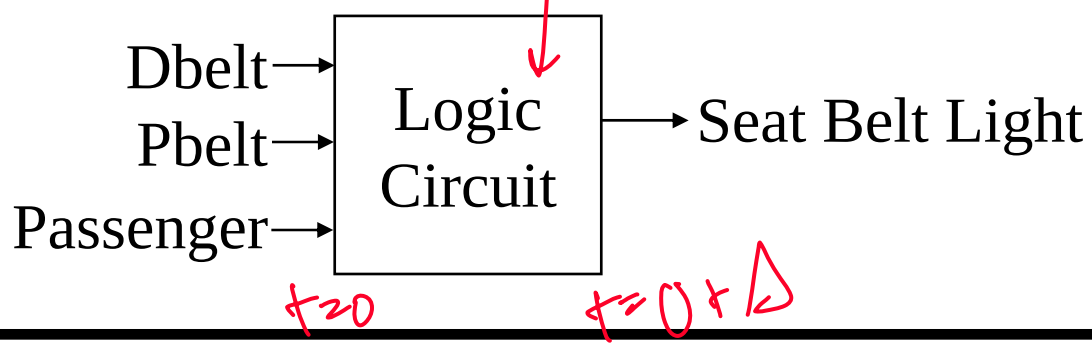
Combinational vs. Sequential Logic



Network implemented from logic gates. The presence of feedback distinguishes between ***sequential*** and ***combinational*** networks.

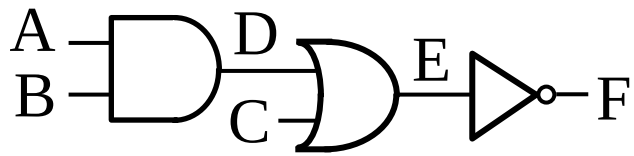
Combinational logic

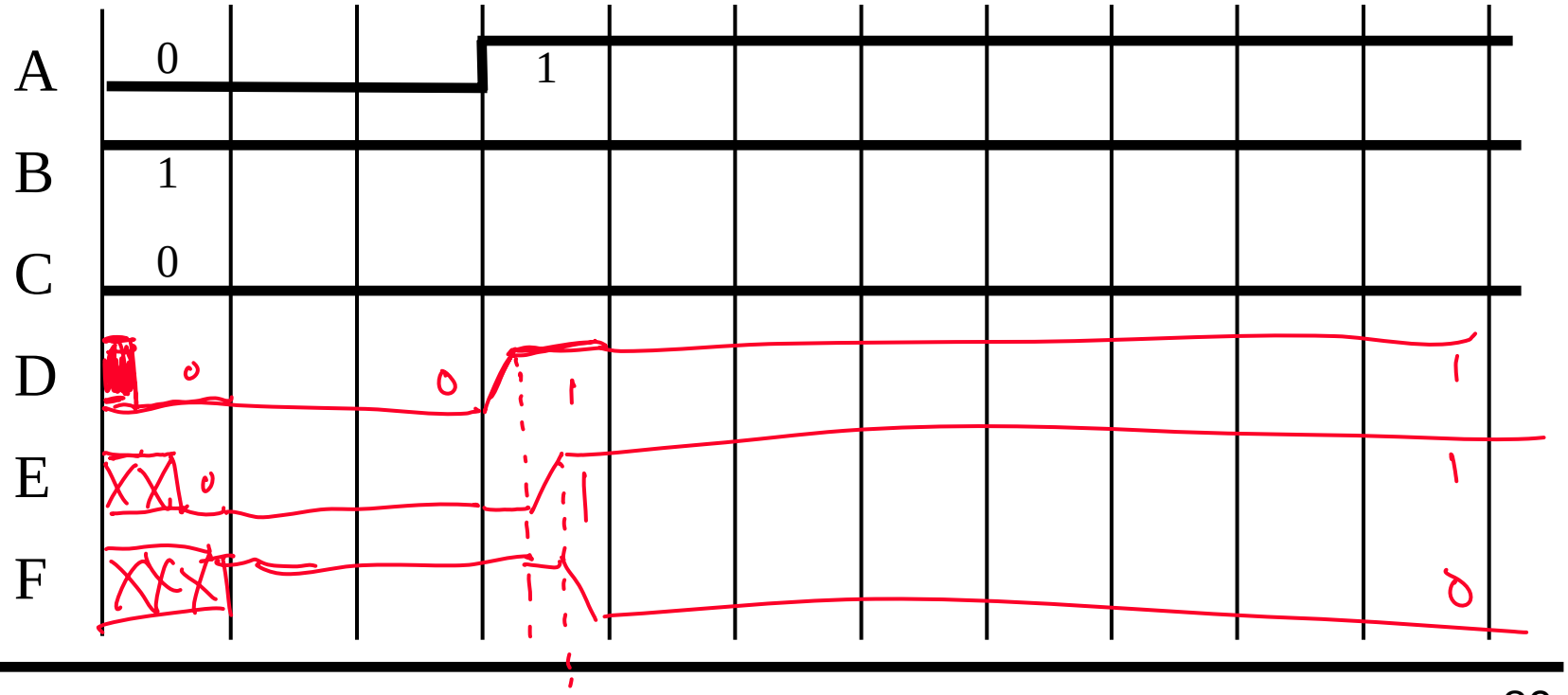
no feedback among inputs and outputs
outputs are a pure function of the inputs
e.g., seat belt light:
(Dbelt, Pbelt, Passenger) mapped into (Light)



Circuit Timing Behavior

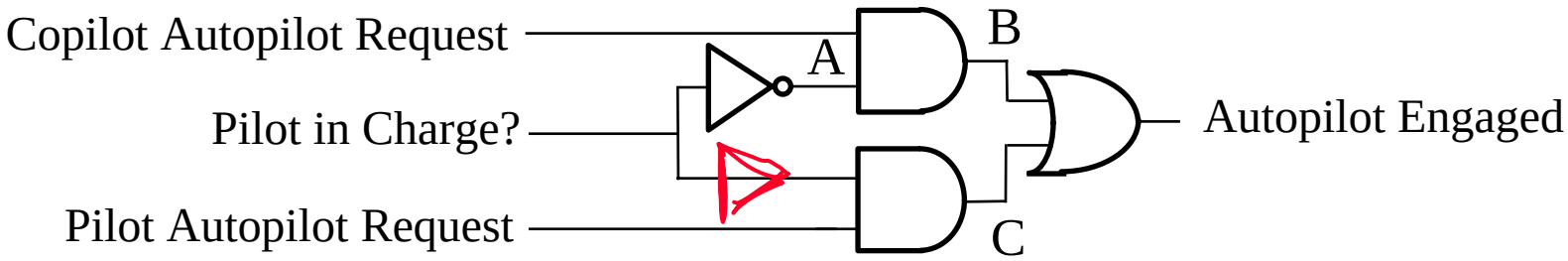
Simple model: gates react after fixed delay

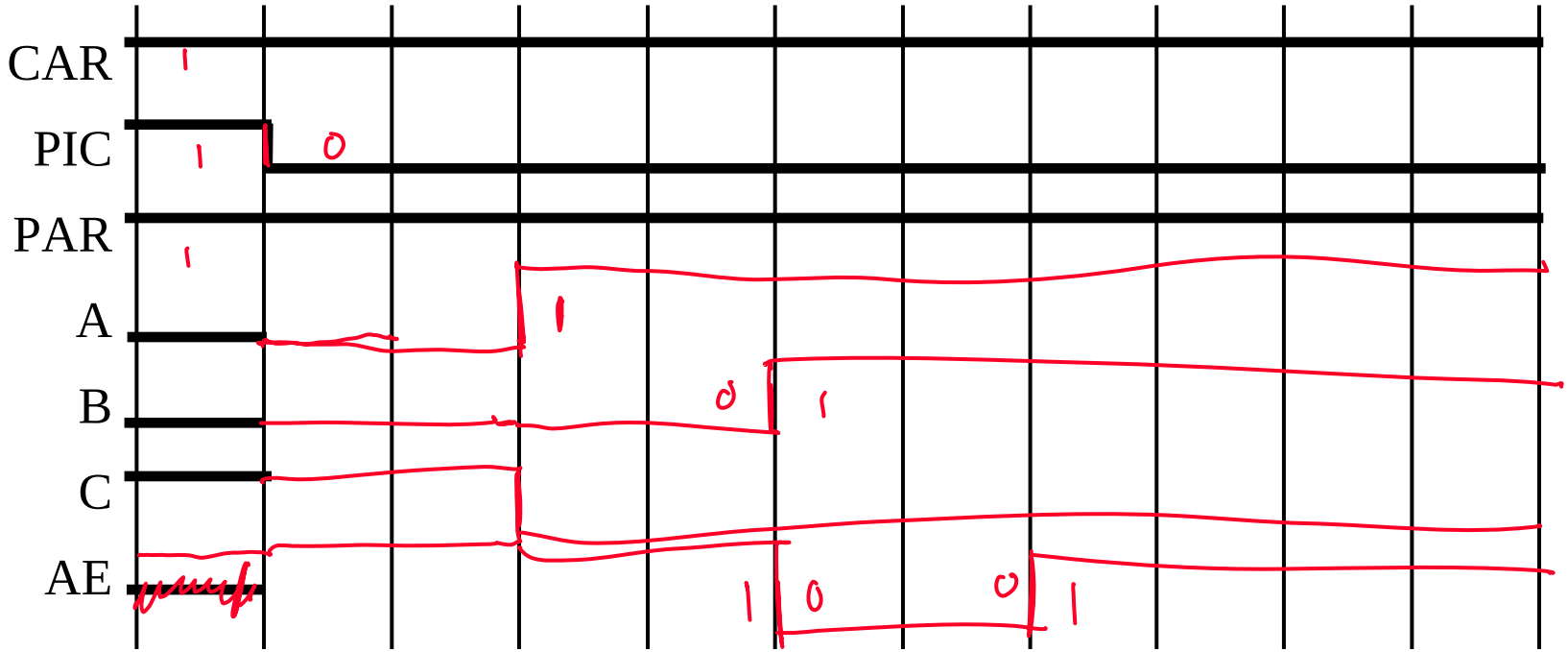




Hazards/Glitches

Circuit can temporarily go to incorrect states



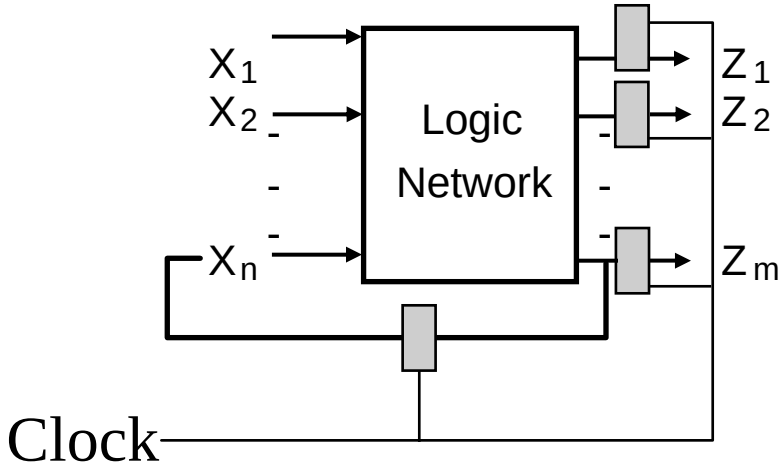


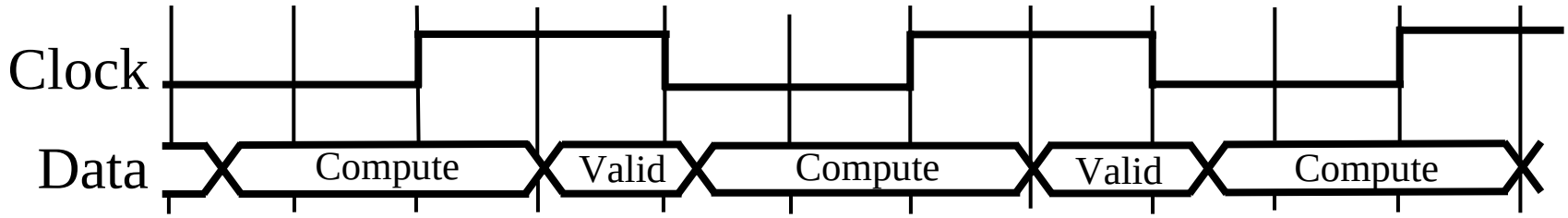
Must filter out temporary states

Safe Sequential Circuits

Clocked elements on feedback, perhaps outputs

- Clock signal synchronizes operation
- Clocked elements hide glitches/hazards





Assembly Language

Readings: Chapter 2 (2.1-2.6, 2.8, 2.9, 2.13, 2.15), Appendix A.10

Assembly language

- Simple, regular instructions – building blocks of C & other languages
- Typically one-to-one mapping to machine language

Our goal

- Understand the basics of assembly language
- Help figure out what the processor needs to be able to do

<u>Not</u> our goal to teach complete assembly/machine language programming

- Floating point
- Procedure calls
- Stacks & local variables

MIPS Assembly Language

The basic instructions have four components:

- Operator name
- Destination
- 1st operand
- 2nd operand

```
add <dst>, <src1>, <src2>      # <dst> = <src1> + <src2>
sub <dst>, <src1>, <src2>      # <dst> = <src1> - <src2>
```

Simple format: easy to implement in hardware

More complex: `A = B + C + D - E`

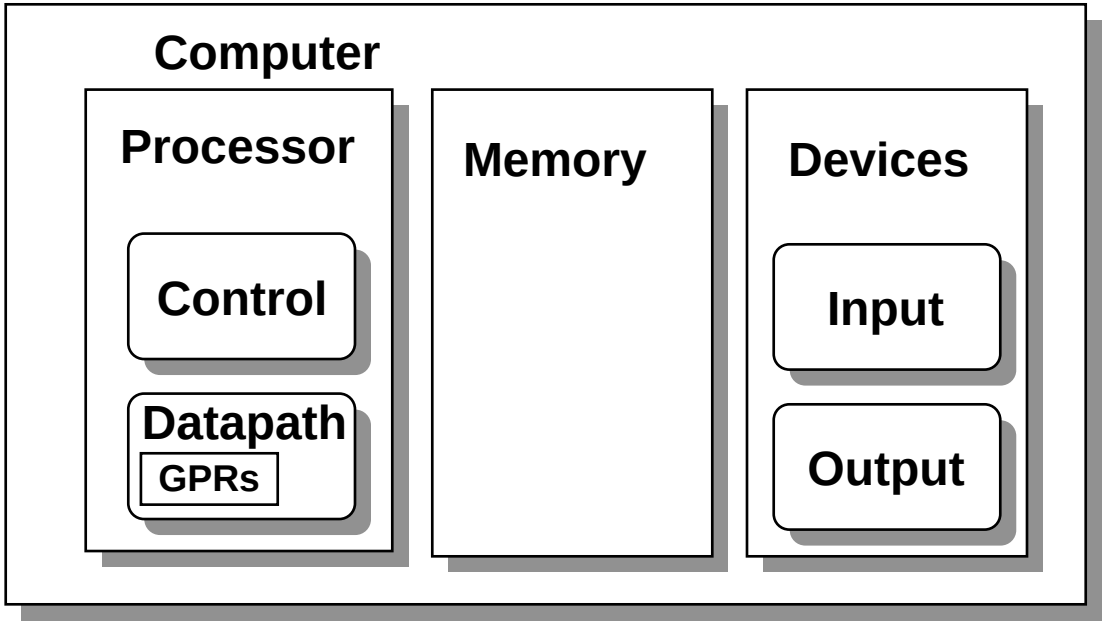
t_1

t_2

A

Operands & Storage

For speed, CPU has 32 general-purpose registers for storing most operands
For capacity, computer has large memory (64MB+)



Load/store operation moves information between registers and main memory
All other operations work on registers

Registers

32 registers for operands

Register	Name	Function	Comment
$0	$zero	Always 0	No-op on write
$1	$at	Reserved for assembler	Don’t use it!
$2-3	$v0-v1	Function return	
$4-7	$a0-a3	Function call parameters	
$8-15	$t0-t7	Volatile temporaries	Not saved on call
$16-23	$s0-s7	Temporaries (saved across calls)	Saved on call
$24-25	$t8-t9	Volatile temporaries	Not saved on call
$26-27	$k0-k1	Reserved kernel/OS	Don’t use them
$28	$gp	Pointer to global data area	
$29	$sp	Stack pointer	
$30	$fp	Frame pointer	
$31	$ra	Function return address	

Basic Operations

(Note: just subset of all instructions)

Mathematic: add, sub, mult, div	`add $t0, $t1, $t2 # t0 = t1+t2`
Unsigned (changes overflow condition)	`addu $t0, $t1, $t2 # t0 = t1+t2`
Immediate (one input a constant)	`addi $t0, $t1, 100 # t0 = t1+100`
Logical: and, or, nor, xor	`and $t0, $t1, $t2 # t0 = t1&t2`
Immediate	`andi $t0, $t1, 7 # t0 = t1&b0111`
Shift: left & right logical, arithmetic	`sllv $t0, $t1, $t2 # t0 = t1<<t2`
Immediate	`sll $t0, $t1, 6 # t0 = t1<<6`

Example: Take bits 6-4 of $t0 and make them bits 2-0 of $t1, zeros otherwise:

srli $tK, $t0, 4
andi $t1, $tK, $7_10

·xxx, ····
0000, ·xxx
0000 0111