N26F300
VLSI SYSTEM DESIGN
(GRADUATE LEVEL)

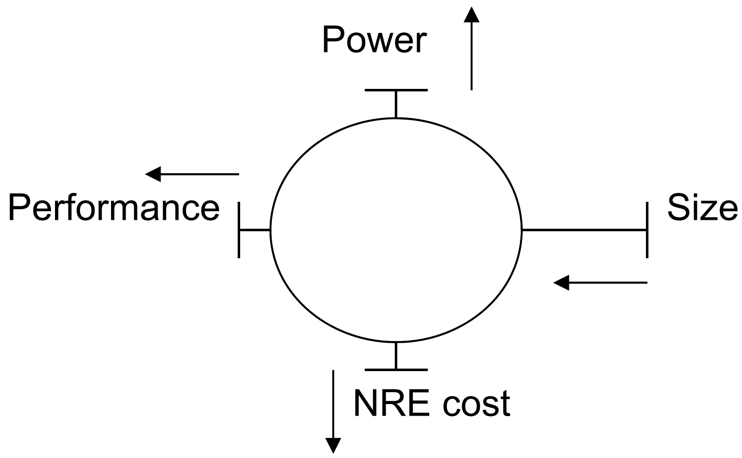
Processors and Peripherals

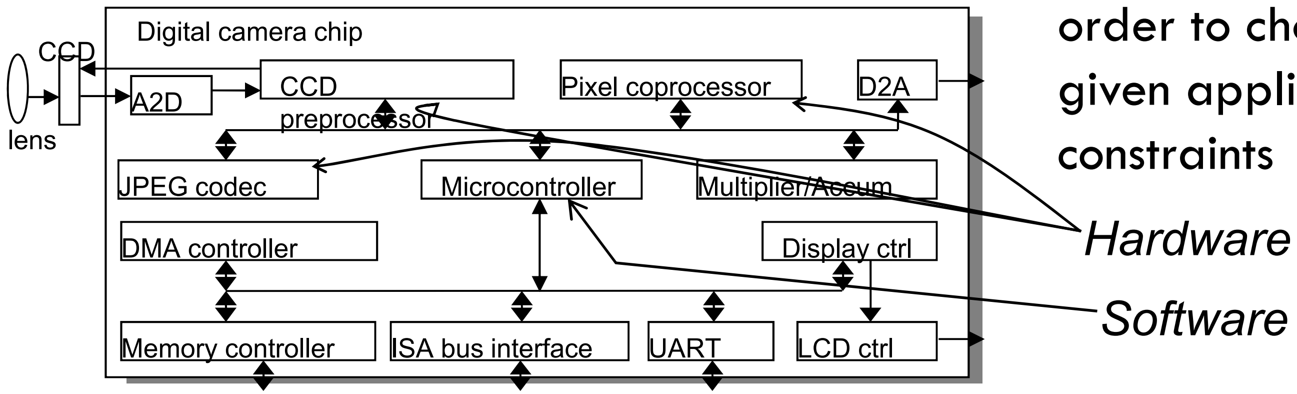
Outline

- Processor
- Custom processor – GCD example
- Peripherals

[Material partly adapted from Embedded System Design by F. Vahid & T. Givargis]

Design metric competition -- improving one may worsen others

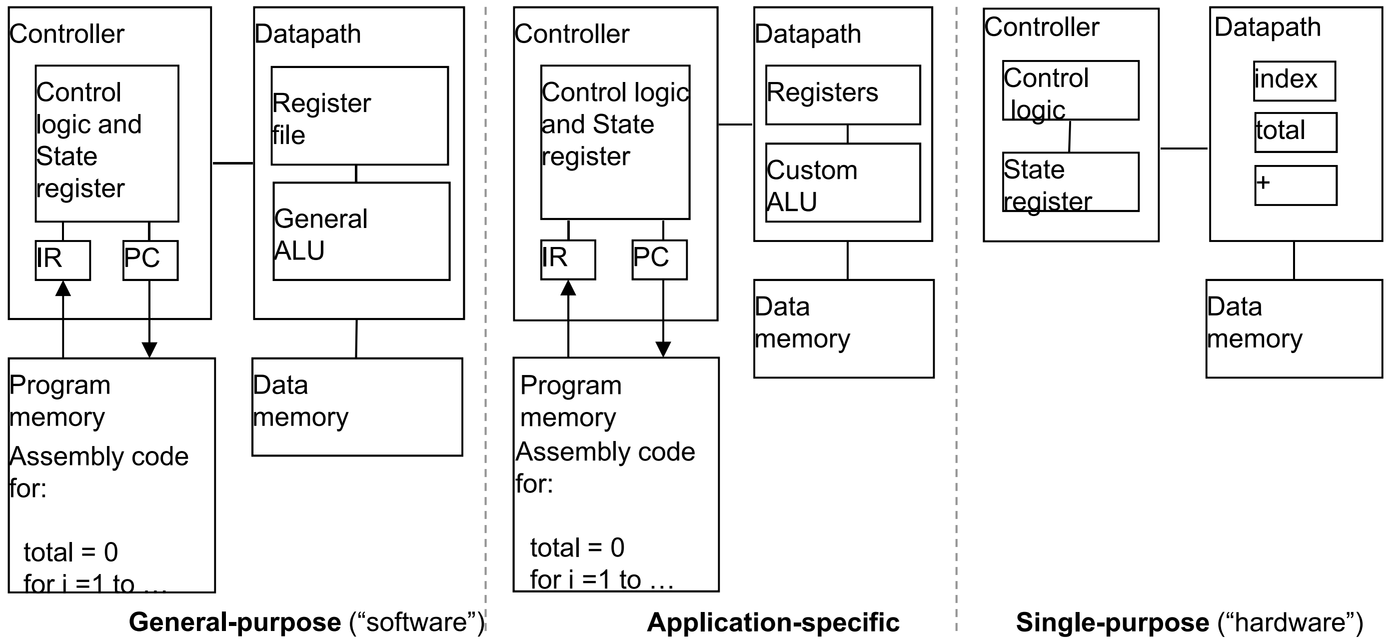




- Expertise with both **software and hardware** is needed to optimize design metrics
  - Not just a hardware or software expert, as is common
  - A designer must be comfortable with various technologies in order to choose the best for a given application and constraints

Processor technology

- The architecture of the computation engine used to implement a system's desired functionality
- Processor does not have to be programmable
  - "Processor" *not* equal to general-purpose processor



Processor technology

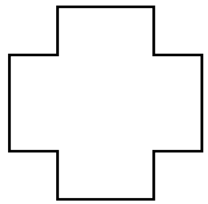
- Processors vary in their customization for the problem at hand

```
total = 0
for i = 1 to N  loop
   total += M[i]
end loop
```

Desired functionality

General-purpose processor

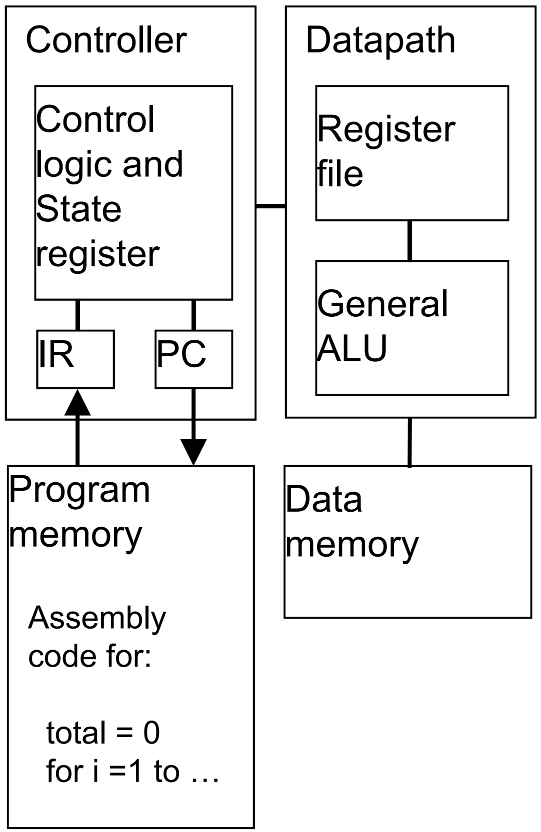
Application-specific processor



Single-purpose processor

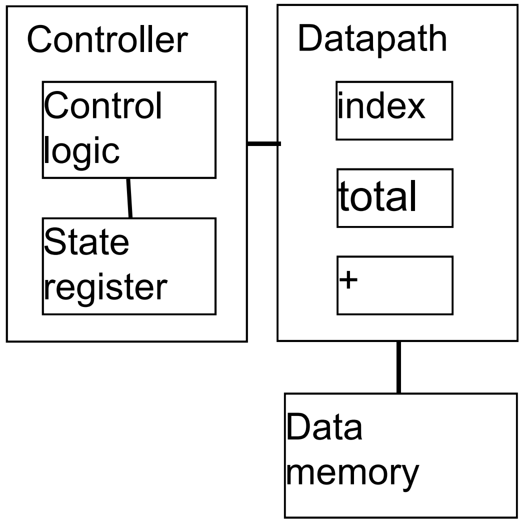
General-purpose processors

- Programmable device used in a variety of applications
  - Also known as "microprocessor"
- Features
  - Program memory
  - General datapath with large register file and general ALU
- User benefits
  - Low time-to-market and NRE costs
  - High flexibility
- "Pentium" the most well-known, but there are hundreds of others



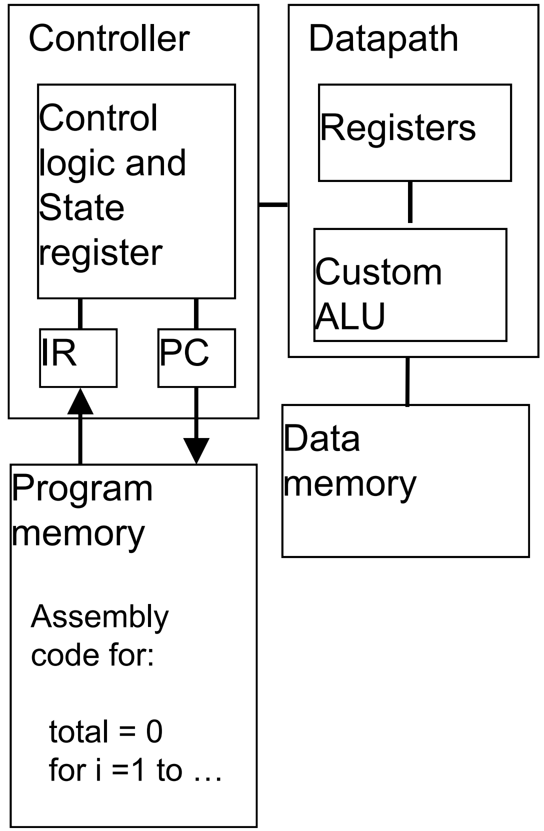
Single-purpose processors

- Digital circuit designed to execute exactly one program
  - a.k.a. coprocessor, accelerator or peripheral
- Features
  - Contains only the components needed to execute a single program
  - No program memory
- Benefits
  - Fast
  - Low power
  - Small size



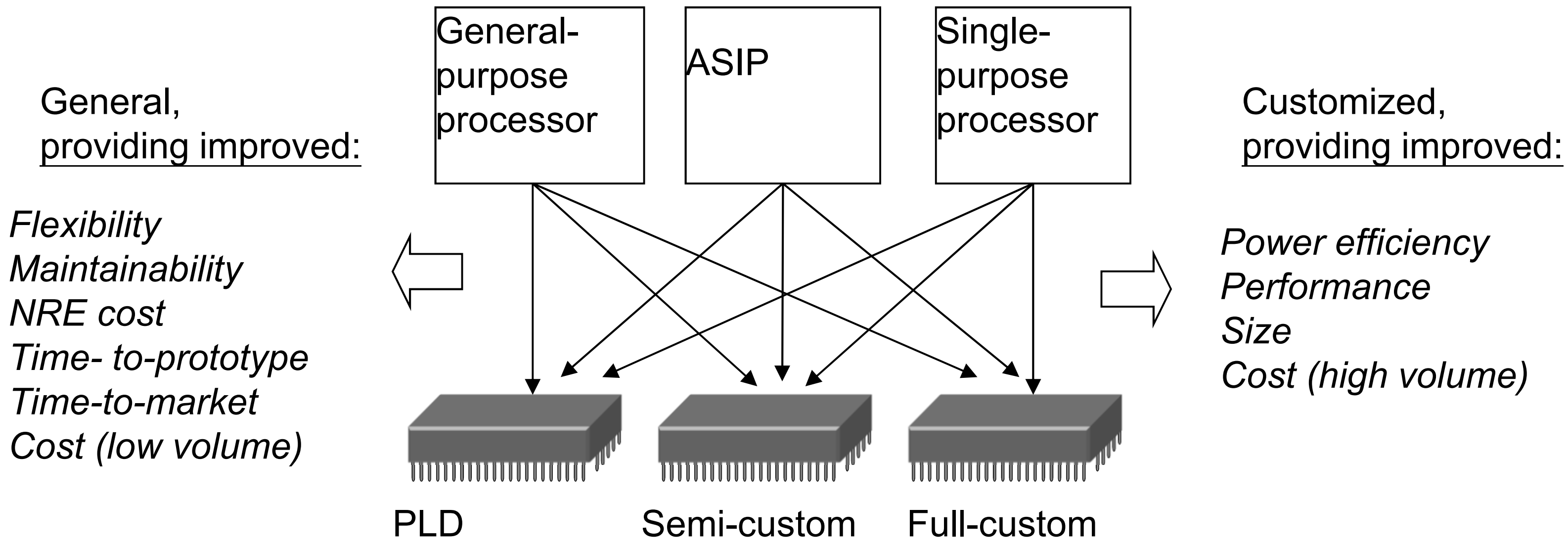
Application-specific processors

- Programmable processor optimized for a particular class of applications having common characteristics
  - Compromise between general-purpose and single-purpose processors
- Features
  - Program memory
  - Optimized datapath
  - Special functional units
- Benefits
  - Some flexibility, good performance, size and power



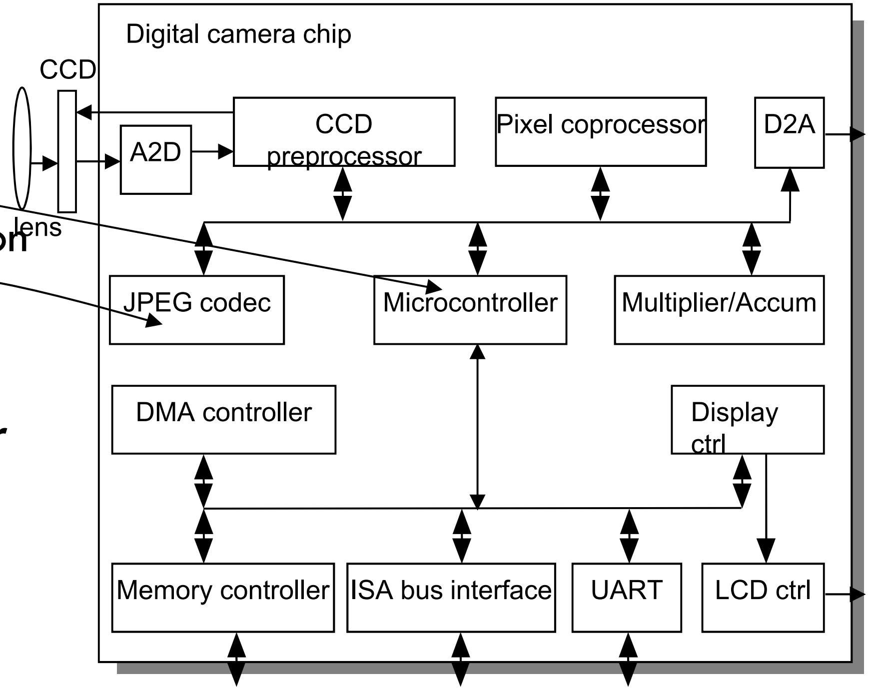
Independence of processor and IC technologies

- Basic tradeoff
  - General vs. custom
  - With respect to processor technology or IC technology
  - The two technologies are independent



Custom Processor

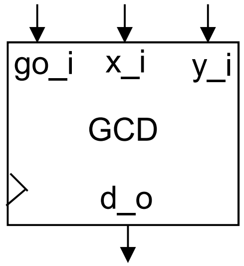
- Processor
  - Digital circuit that performs a computation tasks
  - Controller and datapath
  - General-purpose: variety of computation tasks
  - Single-purpose: one particular computation task
  - Custom single-purpose: non-standard task
- A custom single-purpose processor may be
  - Fast, small, low power
  - But, high NRE, longer time-to-market, less flexible



Example: greatest common divisor

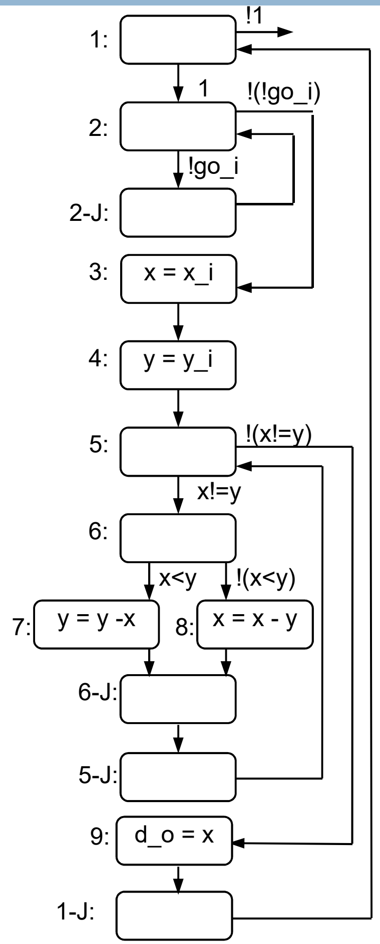
- First create algorithm
- Convert algorithm to "complex" state machine
  - Known as FSMD: finite-state machine with datapath
  - Can use templates to perform such conversion

(a) black-box view



(b) desired functionality

```
0: int  x, y;
1: while (1) {
2:   while (!go_i);
3:   x = x_i;
4:   y = y_i;
5:   while  (x != y)  {
6:      if  (x < y)
7:          y = y - x;
        else
8:          x = x - y;
     }
9:   d_o = x;
   }
```

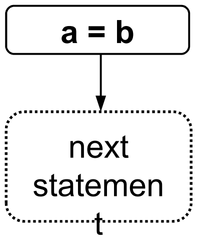


(c) state diagram

State diagram templates

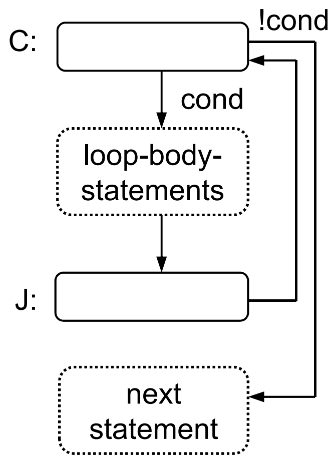
Assignment statement

a = b
next
statement



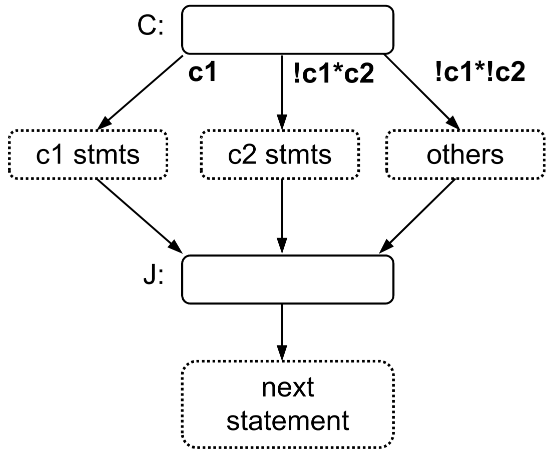
Loop statement

while (cond) {
loop-body-
statements
}
next statement



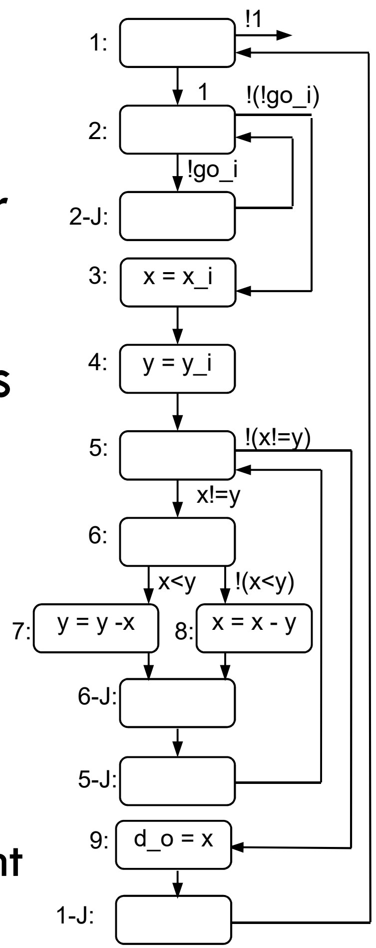
Branch statement

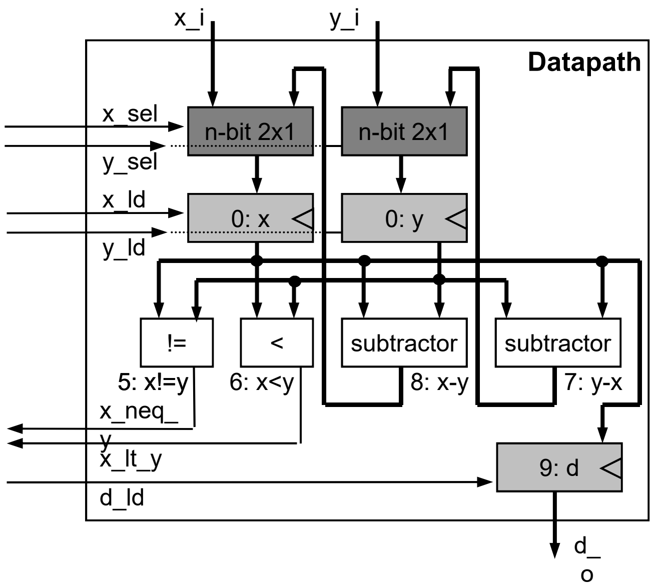
if (c1)
c1 stmts
else if c2
c2 stmts
else
other stmts
next statement



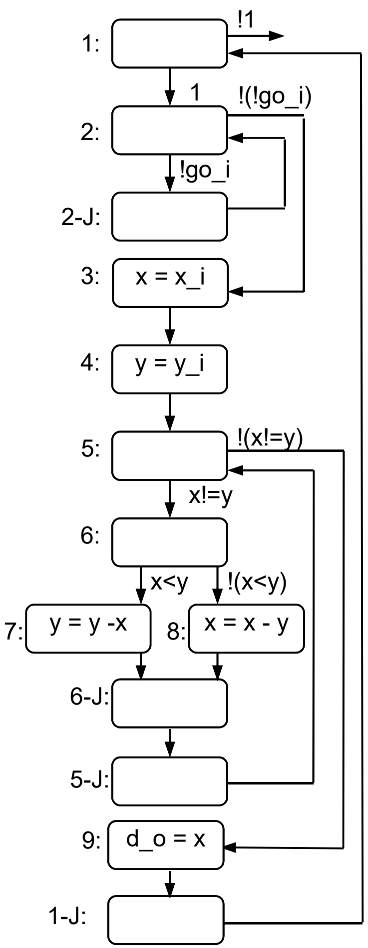
Creating the datapath

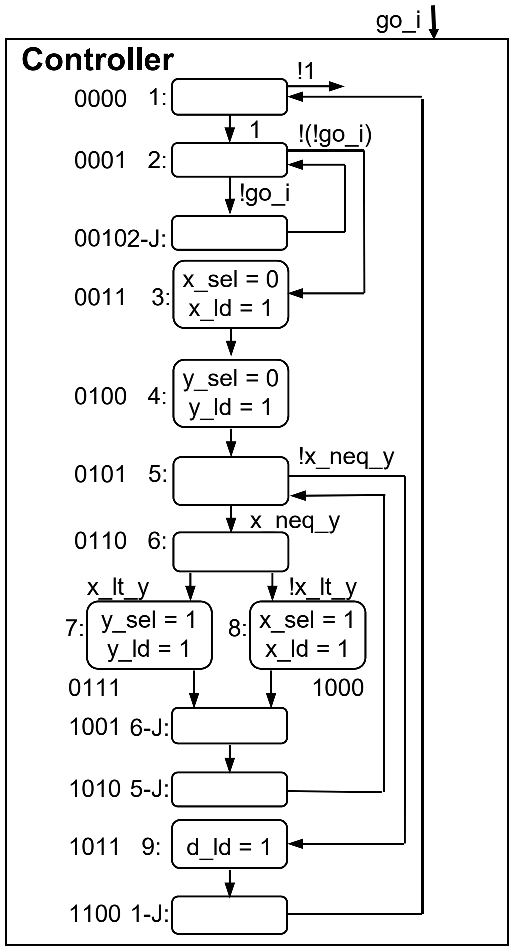
- Create a register for any declared variable
- Create a functional unit for each arithmetic operation
- Connect the ports, registers and functional units
  - Based on reads and writes
  - Use multiplexors for multiple sources
- Create unique identifier
  - for each datapath component control input and output



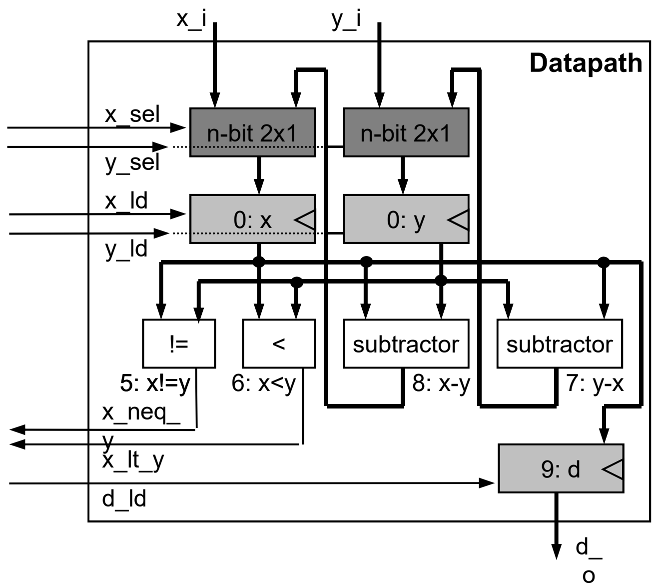


Creating the controller's FSM

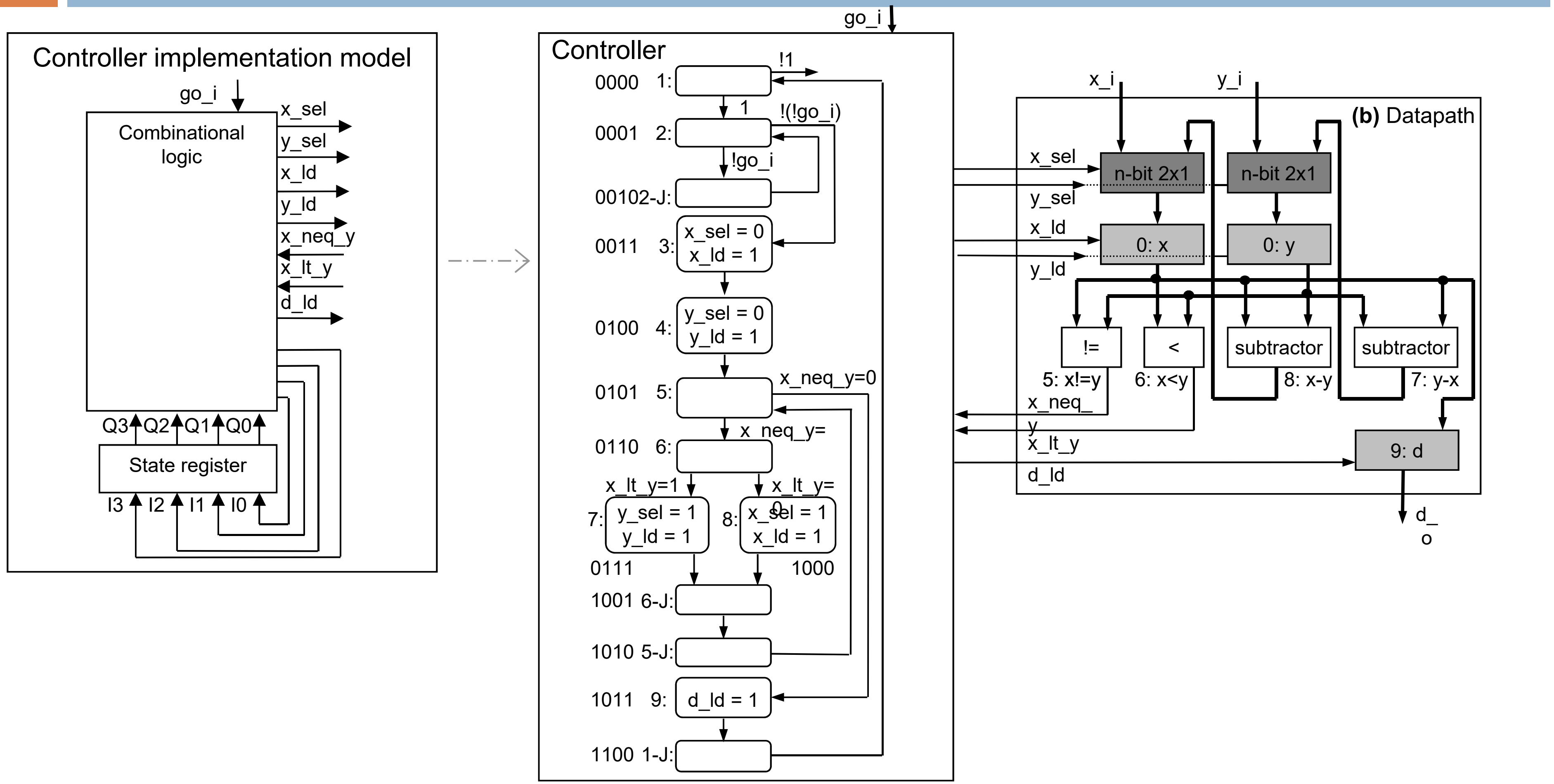




- Same structure as FSMD
- Replace complex actions/conditions with datapath configurations



Splitting into a controller and datapath

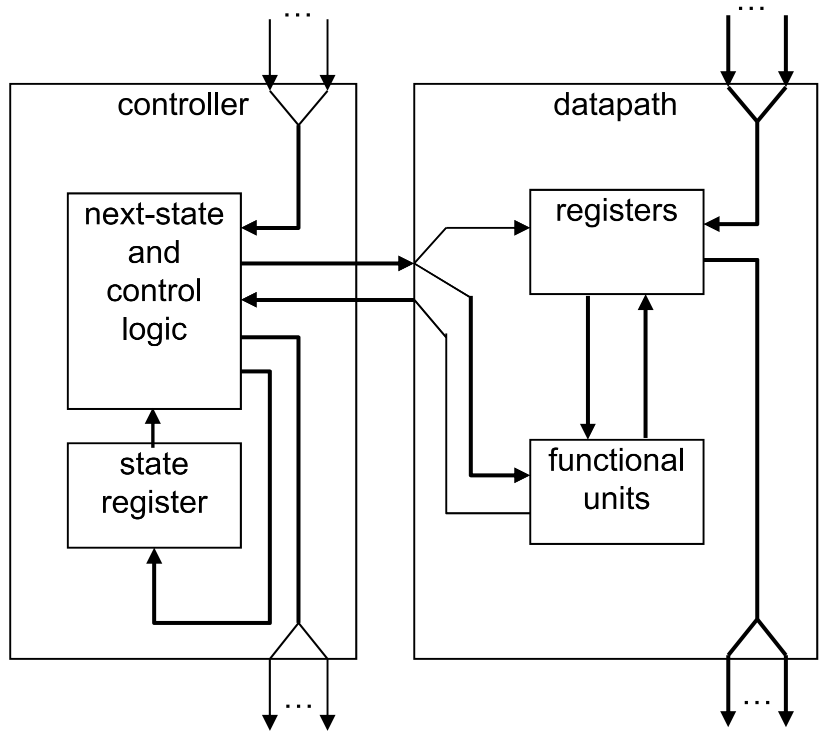


Controller state table for the GCD example

Inputs							Outputs									
Q3	Q2	Q1	Q0	x_neq_y	x_lt_y	go_i	I3	I2	I1	I0	x_sel	y_sel	x_ld	y_ld	d_ld
0	0	0	0	*	*	*	0	0	0	1	X	X	0	0	0
0	0	0	1	*	*	0	0	0	1	0	X	X	0	0	0
0	0	0	1	*	*	1	0	0	1	1	X	X	0	0	0
0	0	1	0	*	*	*	0	0	0	1	X	X	0	0	0
0	0	1	1	*	*	*	0	1	0	0	0	X	1	0	0
0	1	0	0	*	*	*	0	1	0	1	X	0	0	1	0
0	1	0	1	0	*	*	1	0	1	1	X	X	0	0	0
0	1	0	1	1	*	*	0	1	1	0	X	X	0	0	0
0	1	1	0	*	0	*	1	0	0	0	X	X	0	0	0
0	1	1	0	*	1	*	0	1	1	1	X	X	0	0	0
0	1	1	1	*	*	*	1	0	0	1	X	1	0	1	0
1	0	0	0	*	*	*	1	0	0	1	1	X	1	0	0
1	0	0	1	*	*	*	1	0	1	0	X	X	0	0	0
1	0	1	0	*	*	*	0	1	0	1	X	X	0	0	0
1	0	1	1	*	*	*	1	1	0	0	X	X	0	0	1
1	1	0	0	*	*	*	0	0	0	0	X	X	0	0	0
1	1	0	1	*	*	*	0	0	0	0	X	X	0	0	0
1	1	1	0	*	*	*	0	0	0	0	X	X	0	0	0
1	1	1	1	*	*	*	0	0	0	0	X	X	0	0	0

Completing the GCD custom single-purpose processor design

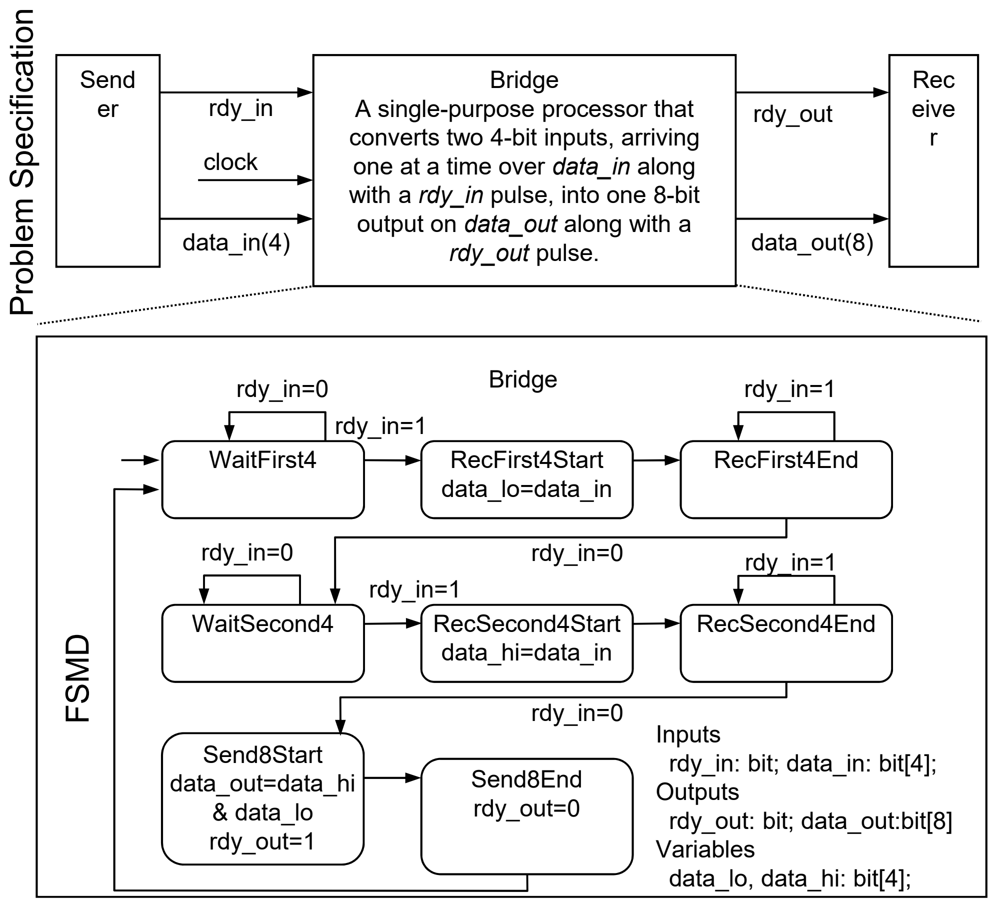
- We finished the datapath
- We have a state table for the next state and control logic
  - All that's left is combinational logic design
- This is *not* an optimized design, but we see the basic steps



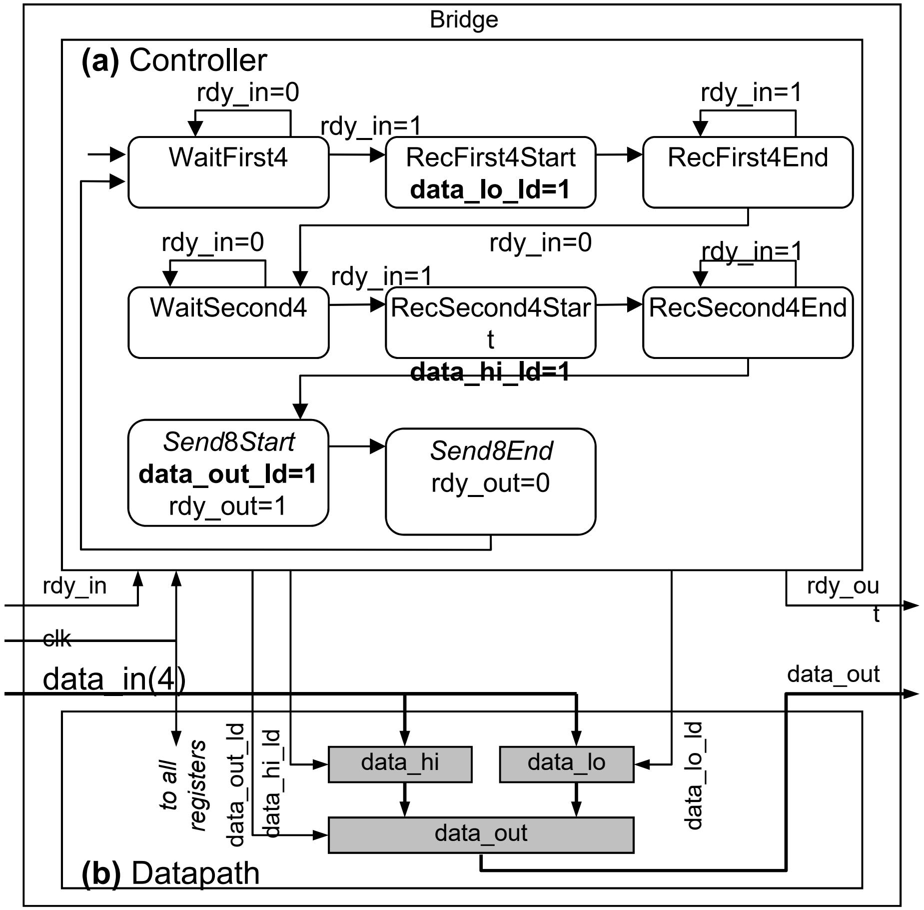
a view inside the controller and datapath

RT-level custom single-purpose processor design

- We often start with a state machine
  - Rather than algorithm
  - Cycle timing often too central to functionality
- Example
  - Bus bridge that converts 4-bit bus to 8-bit bus
  - Start with FSMD
  - Known as register-transfer (RT) level
  - Exercise: complete the design



RT-level custom single-purpose processor design (cont’)



Optimizing single-purpose processors

- Optimization is the task of making design metric values the best possible
- Optimization opportunities
  - original program
  - FSMD
  - datapath
  - FSM

Optimizing the original program

- Analyze program attributes and look for areas of possible improvement
  - number of computations
  - size of variable
  - time and space complexity
  - operations used
    - multiplication and division very expensive

Optimizing the original program (cont')

original program

```
0: int  x, y;
1: while (1) {
2:   while (!go_i);
3:   x = x_i;
4:   y = y_i;
5:   while  (x != y)  {
6:      if  (x < y)
7:         y = y - x;
        else
8:         x = x - y;
     }
9:   d_o = x;
   }
```

replace the subtraction operation(s) with modulo operation in order to speed up program

optimized program

```
0: int  x, y, r;
1: while (1) {
2:    while (!go_i);
      // x must be the larger number
3:    if (x_i >= y_i)  {
4:       x=x_i;
5:       y=y_i;
      }
6:    else {
7:       x=y_i;
8:       y=x_i;
      }
9:    while  (y != 0)  {
10:      r = x % y;
11:      x = y;
12:      y = r;
      }
13:   d_o = x;
   }
```

GCD(42, 8) - 9 iterations to complete the loop
x and y values evaluated as follows : (42, 8), (43, 8), (26,8), (18,8), (10, 8), (2,8), (2,6), (2,4), (2,2).

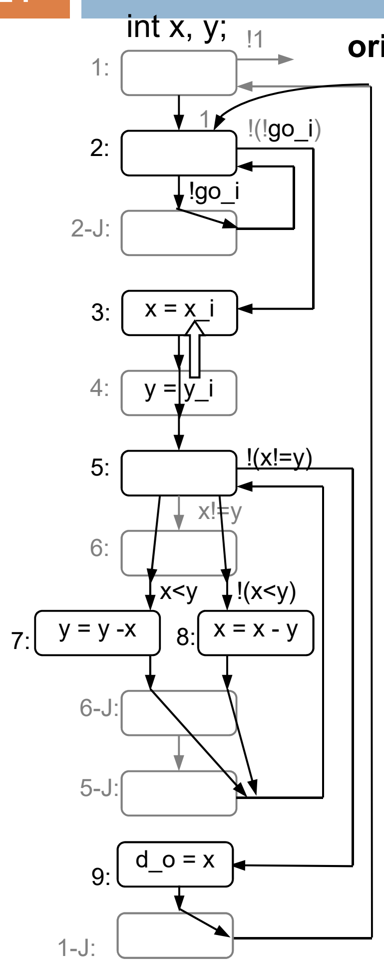
GCD(42,8) - 3 iterations to complete the loop
x and y values evaluated as follows: (42, 8), (8,2), (2,0)

Optimizing the FSMD

- Areas of possible improvements
  - merge states
    - states with constants on transitions can be eliminated, transition taken is already known
    - states with independent operations can be merged
  - separate states
    - states which require complex operations (a*b*c*d) can be broken into smaller states to reduce hardware size
  - scheduling

Optimizing the FSMD (cont.)

original FSMD



eliminate state 1 – transitions have constant values

merge state 2 and state 2J – no loop operation in between them

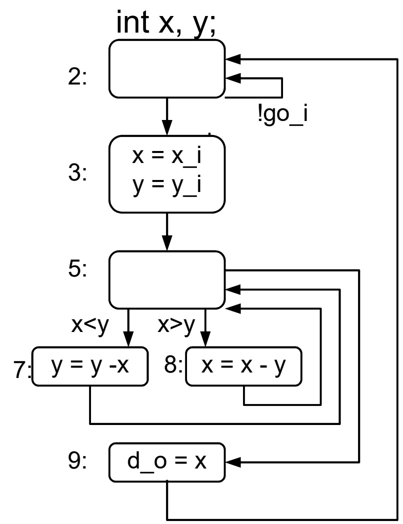
merge state 3 and state 4 – assignment operations are independent of one another

merge state 5 and state 6 – transitions from state 6 can be done in state 5

eliminate state 5J and 6J – transitions from each state can be done from state 7 and state 8, respectively

eliminate state 1-J – transition from state 1-J can be done directly from state 9

optimized FSMD



Optimizing the datapath

- Sharing of functional units
  - one-to-one mapping, as done previously, is not necessary
  - if same operation occurs in different states, they can share a single functional unit
- Multi-functional units
  - ALUs support a variety of operations, it can be shared among operations occurring in different states

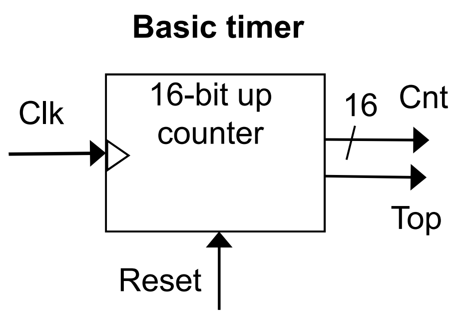
Optimizing the FSM

- State encoding
  - task of assigning a unique bit pattern to each state in an FSM
  - size of state register and combinational logic vary
  - can be treated as an ordering problem
- State minimization
  - task of merging equivalent states into a single state
    - state equivalent if for all possible input combinations the two states generate the same outputs and transitions to the next same state

Peripherals

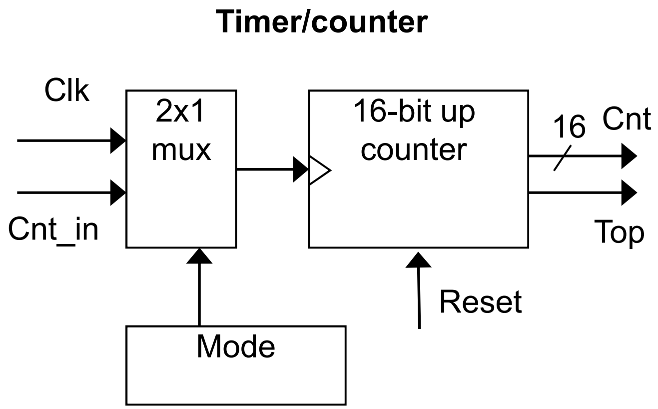
Timers, counters, watchdog timers

- Timer: measures time intervals
  - To generate timed output events
    - e.g., hold traffic light green for 10 s
  - To measure input events
    - e.g., measure a car's speed
- Based on counting clock pulses
    - E.g., let Clk period be 10 ns
    - And we count 20,000 Clk pulses
    - Then 200 microseconds have passed
    - 16-bit counter would count up to 65,535*10 ns = 655.35 microsec., resolution = 10 ns
    - Top: indicates top count reached, wrap-around



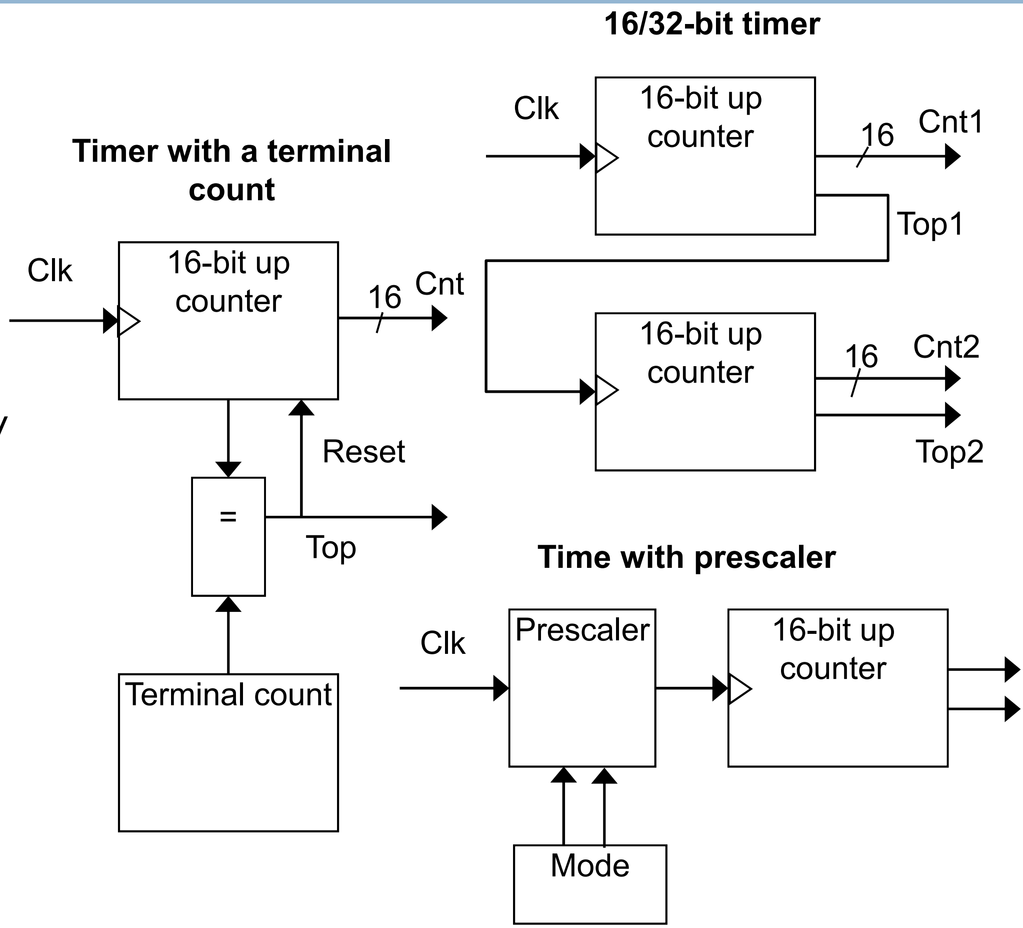
Counters

- Counter: like a timer, but counts pulses on a general input signal rather than clock
  - e.g., count cars passing over a sensor
  - Can often configure device as either a timer or counter

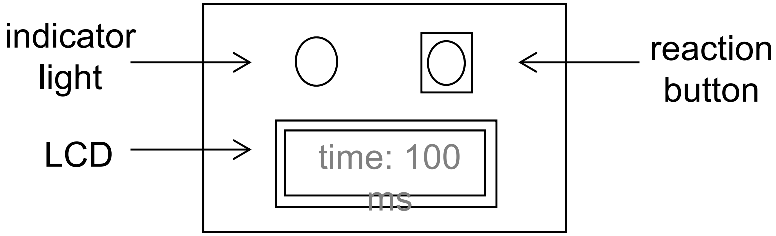


Other timer structures

- Interval timer
  - Indicates when desired time interval has passed
  - We set terminal count to desired interval
    - *Number of clock cycles = Desired time interval / Clock period*
- Cascaded counters
- Prescaler
  - Divides clock
  - Increases range, decreases resolution



Example: Reaction Timer

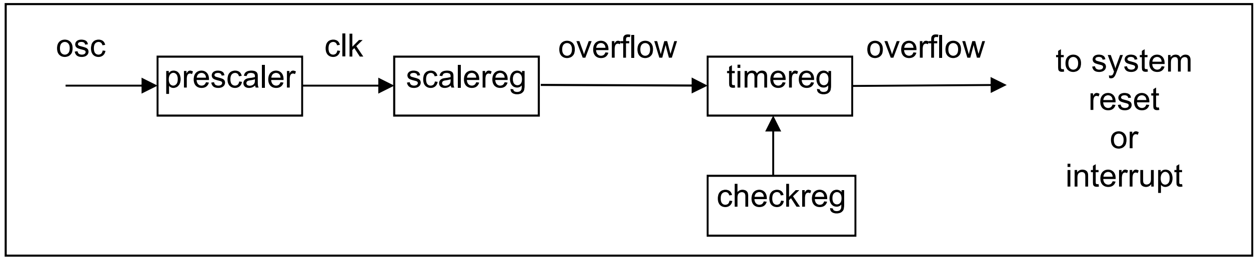


- Measure time between turning light on and user pushing button
  - 16-bit timer, clk period is 83.33 ns, counter increments every 6 cycles
  - Resolution = 6*83.33=0.5 microsec.
  - Range = 65535*0.5 microseconds = 32.77 milliseconds
  - Want program to count millisec., so initialize counter to 65535 – 1000/0.5 = 63535

```
/* main.c */

#define MS_INIT      63535
void main(void){
  int count_milliseconds = 0;

  configure timer mode
  set Cnt to MS_INIT

  wait a random amount of time
  turn on indicator light
  start timer

while (user has not pushed reaction button){
    if(Top) {
      stop timer
      set Cnt to MS_INIT
      start timer
      reset Top
      count_milliseconds++;
    }
  }
  turn light off
  printf("time:  %i ms", count_milliseconds);
}
```

Watchdog timer

- Must reset timer every X time unit, else timer generates a signal
- Common use: detect failure, self-reset
- Another use: timeouts
  - e.g., ATM machine
  - 16-bit timer, 2 microsec. resolution
  - *timereg* value = $2*(2^{16}-1)–X = 131070–X$
  - For 2 min., X = 120,000 microsec.




```
/* main.c */

main(){
  wait until card inserted
  call watchdog_reset_routine

  while(transaction in progress){
    if(button pressed){
      perform corresponding action
      call watchdog_reset_routine
    }

/* if watchdog_reset_routine not called
every < 2 minutes,
interrupt_service_routine is called */
}
```

```
watchdog_reset_routine(){
/* checkreg is set so we can load value
into timereg.  Zero is loaded into
scalereg and 11070 is loaded into
timereg */

  checkreg = 1
  scalereg = 0
  timereg  = 11070
}

void interrupt_service_routine(){
  eject card
  reset screen
}
```

Serial Transmission Using UARTs

- UART: Universal Asynchronous Receiver Transmitter
  - Takes parallel data and transmits serially
  - Receives serial data and converts to parallel
- Parity: extra bit for simple error checking
- Start bit, stop bit
- Baud rate
  - signal changes per second
  - bit rate usually higher