SPECIFICATION AND DESIGN OF EMBEDDED SYSTEMS

by

Daniel D. Gajski
Frank Vahid
Sanjiv Narayan
Jie Gong

University of California at Irvine
Department of Computer Science
Irvine, CA 92715-3425

Design representations

- Behavioral
  Represents functionality but not implementation
- Structural
  Represents connectivity but not dimensionality
- Physical
  Represents dimensionality but not functionality

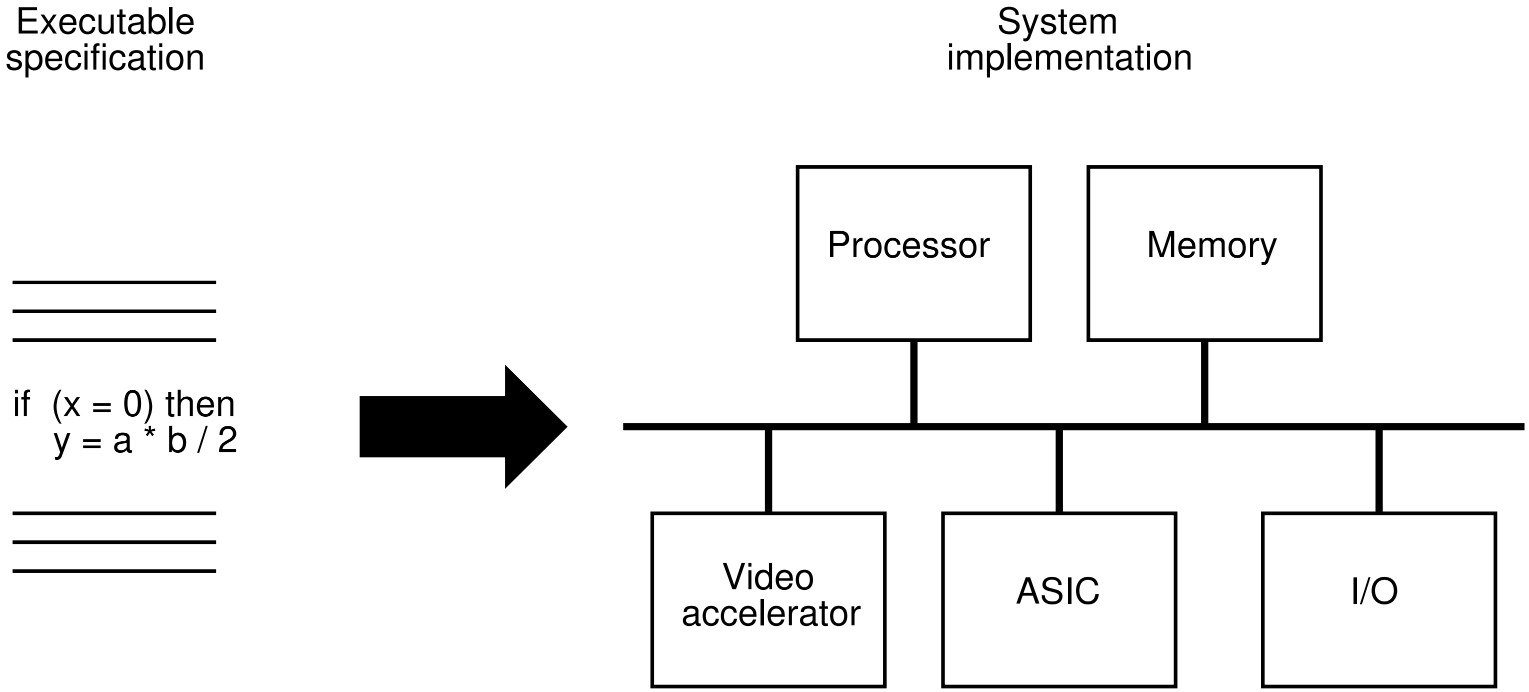
Levels of abstraction

Levels	Behavioral forms	Structural components	Physical objects
Transistor	Differential eq., current−voltage diagrams	Transistors, resistors, capacitors	Analog and digital cells
Gate	Boolean equations, finite−state machines	Gates, flip−flops	Modules, units
Register	Algorithms, flowcharts, instruction sets, generalized FSM	Adders, comparators, registers, counters, register files, queues	Microchips, ASICs
Processor	Executable spec., programs	Processors, controllers, memories, ASICs	PCBs, MCMs

Design methodologies

- Capture-and-simulate
  - Schematic capture
  - Simulation
- Describe-and-synthesize
  - Hardware description language
  - Behavioral synthesis
  - Logic synthesis
- Specify-explore-rene
  - Executable specication
  - Software and hardware partitioning
  - Estimation and exploration
  - Specication renement

Motivation



Models
Languages

Partitioning
Estimation
Refinement

Software compilation
Behavioral synthesis
Logic synthesis

Physical design
Test generation
Manufacturing

Outline

- Introduction
- Design models and architectures
- System-design languages
- An example
- Translation
- Partitioning
- Estimation
- Renement
- Methodology and environments

Models and architectures

Specification + Constraints	**Models** (Specification)
↓ Design process	↓
Implementation	**Architectures** (Implementation)

Models are conceptual views of the system's functionality
Architectures are abstract views of the system's implementation

Models and architectures

- Model: a set of functional objects and rules for composing these objects
- Architecture: a set of implementation components and their connections

Models of an elevator controller

"If the elevator is stationary and the floor requested is equal to the current floor, then the elevator remains idle.

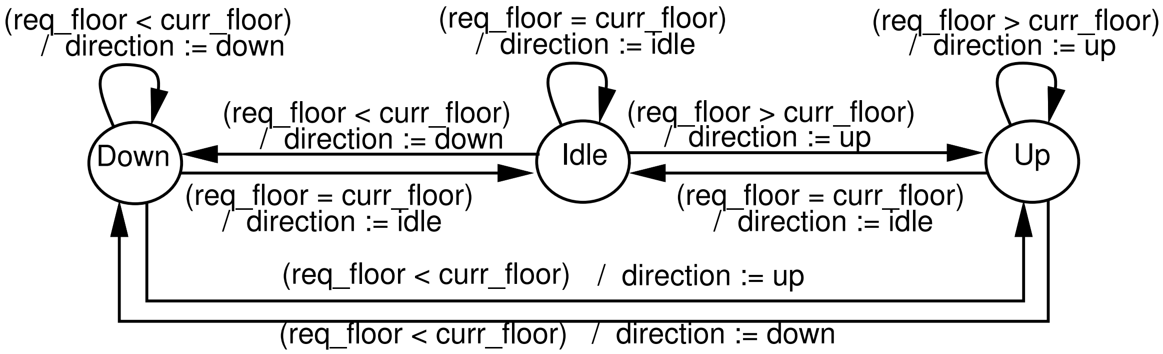
If the elevator is stationary and the floor requested is less than the current floor, then lower the elevator to the requested floor.

If the elevator is stationary and the floor requested is greater than the current floor, then raise the elevator to the requested floor."

(a) English description

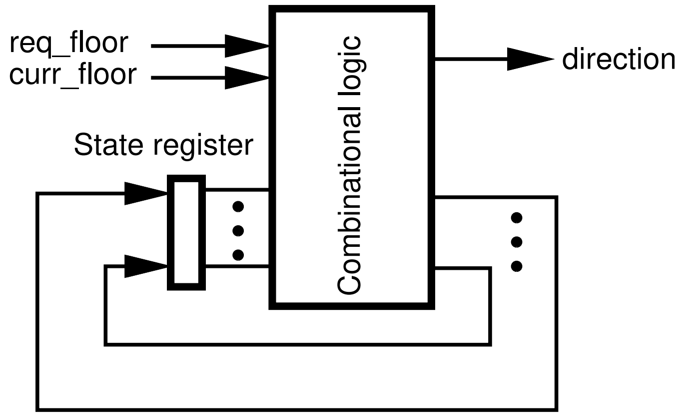
```
loop
    if (req_floor = curr_floor) then
      direction := idle;
    elsif (req_floor < curr_floor) then
      direction := down;
    elsif  (req_floor > curr_floor) then
      direction := up;
    end if;
end loop;
```

(b) Algorithmic model

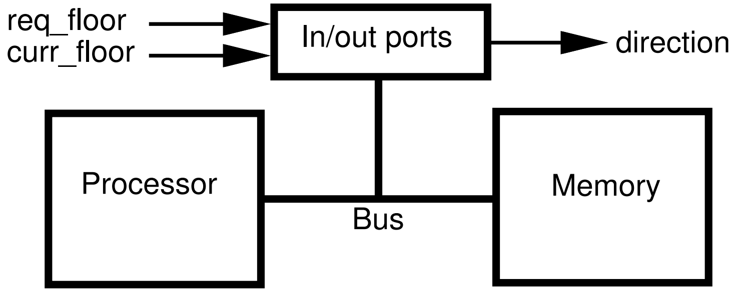


(c) State−machine model

Architectures for implementing the elevator controller



(a) Register level

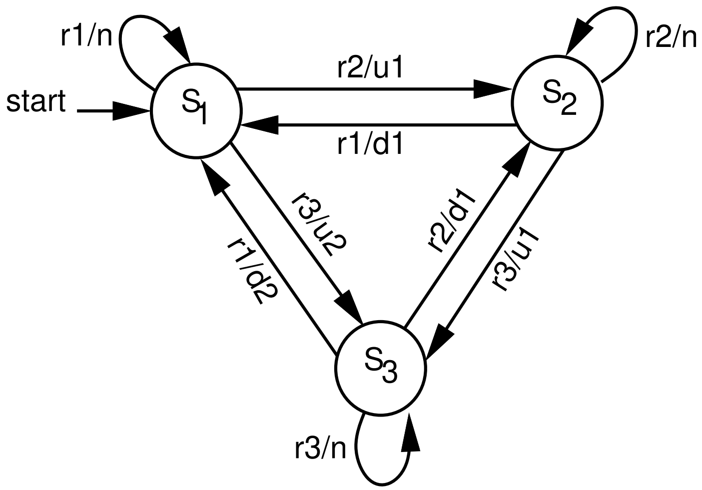


(b) System level

Models

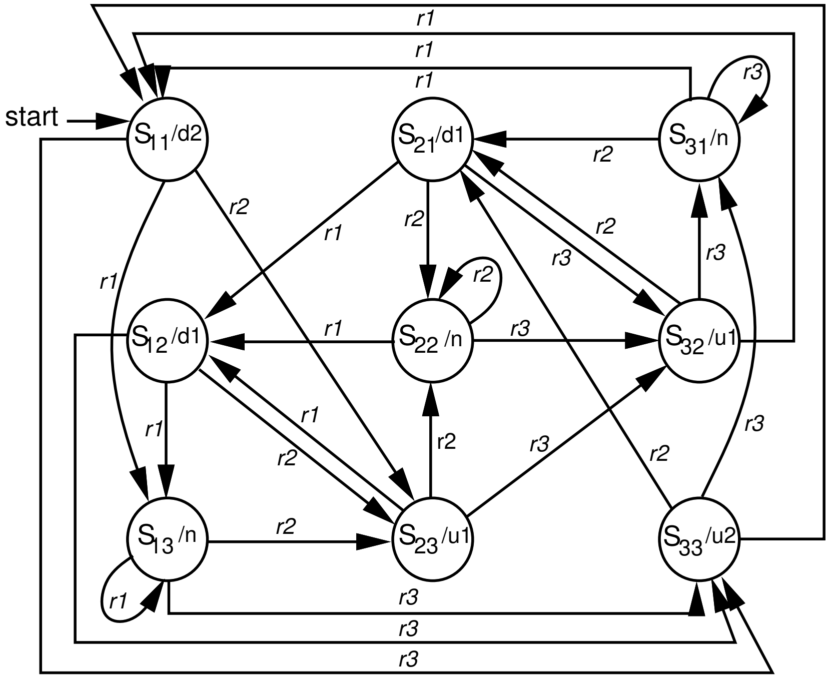
- State-oriented models
  Finite-state machine (FSM), Petri net, Hierarchical concurrent FSM
- Activity-oriented models
  Dataow graph, Flowchart
- Structure-oriented models
  Block diagram, RT netlist, Gate netlist
- Data-oriented models
  Entity-relationship diagram, Jackson's diagram
- Heterogeneous models
  Control/dataow graph, Structure chart, Programming language paradigm, Object-oriented paradigm, Program-state machine, Queueing model

State oriented: Finite-state machine (Mealy model)

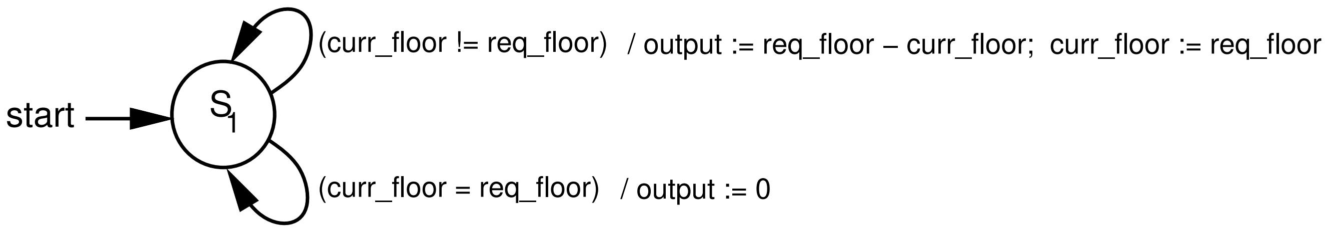


S = { s1, s2, s3}
I = {r1, r2, r3}
O = {d2, d1, n, u1, u2}
f: S x I –> S
h: S x I –> O

State oriented: Finite-state machine (Moore model)



State oriented: Finite-state machine with datapath



Finite-state machines

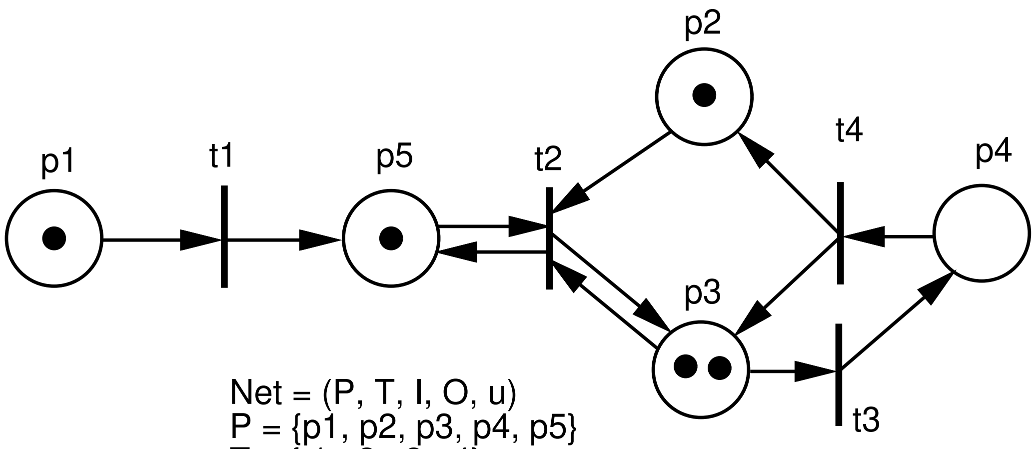
- Merits:

  represent system's temporal behavior explicitly
  suitable for control-dominated system

- Demerits:

  lack of hierarchy and concurrency resulting in
  state or arc explosion when representing complex systems

State oriented: Petri nets



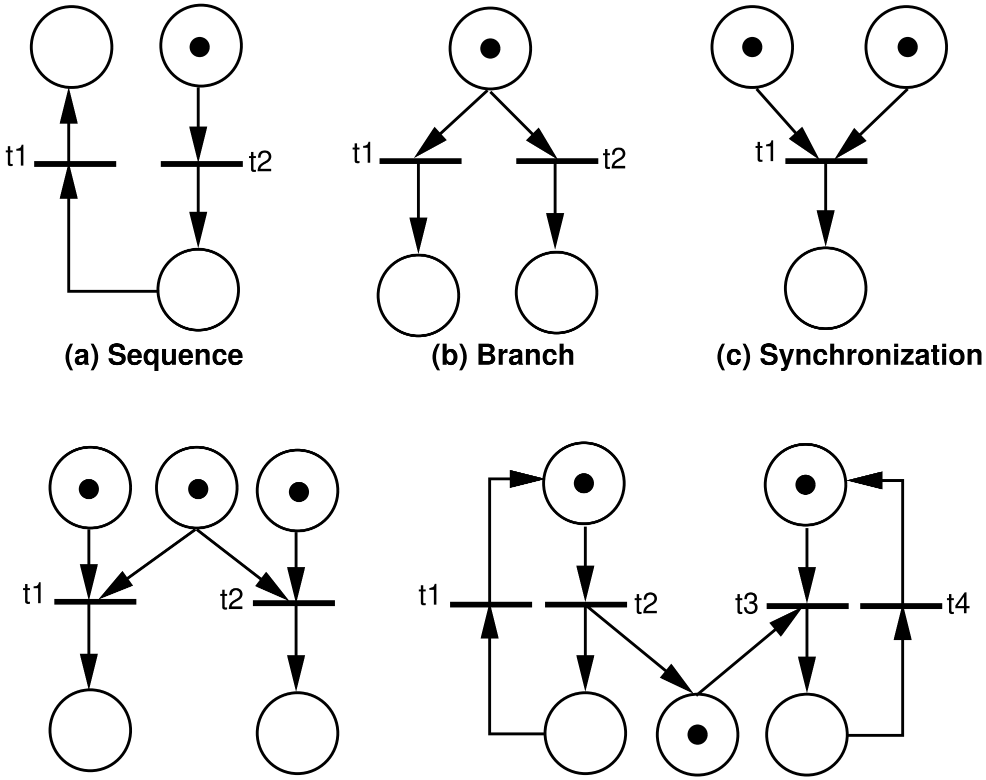
Net = (P, T, I, O, u)
P = {p1, p2, p3, p4, p5}
T = {t1, t2, t3, t4}

I: I(t1) = {p1}
I(t2) = {p2,p3,p5}
I(t3) = {p3}
I(t4) = {p4}

O: O(t1) = {p5}
O(t2) = {p3,p5}
O(t3) = {p4}
O(t4) = {p2,p3}

u: u(p1) = 1
u(p2) = 1
u(p3) = 2
u(p4) = 0
u(p5) = 1

Petri nets



(a) Sequence (b) Branch (c) Synchronization

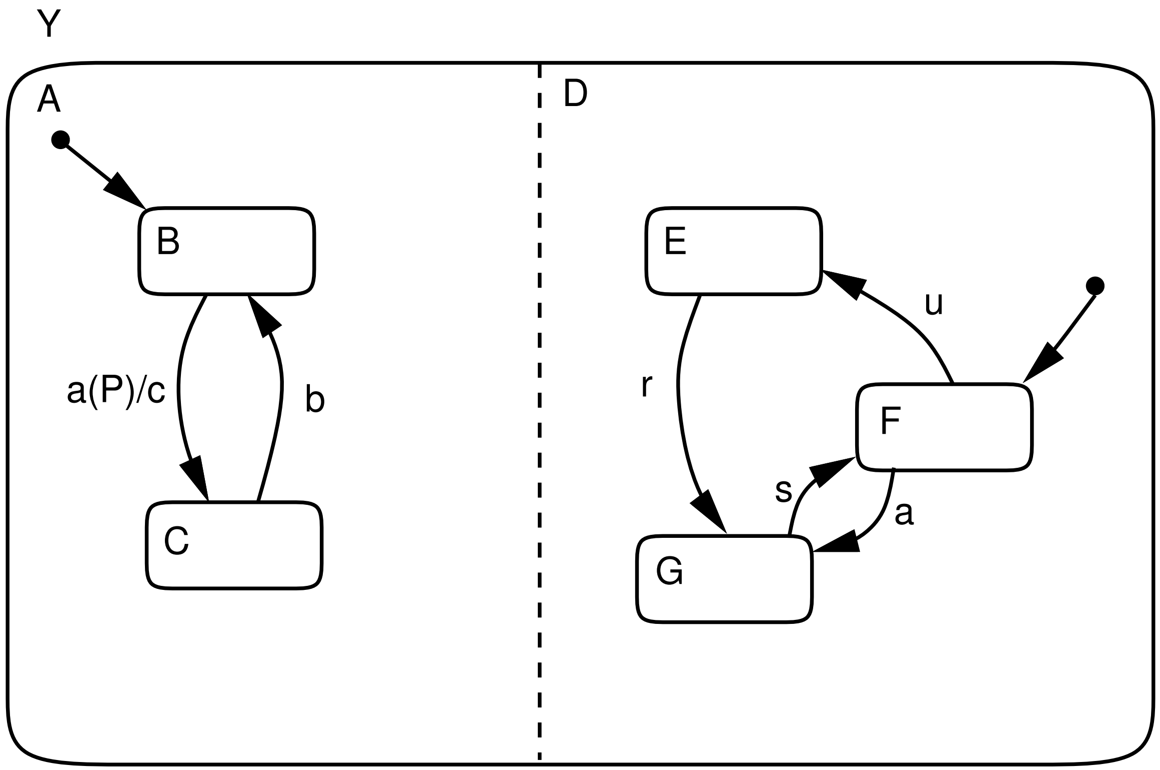
(d) Resource contention (e) Concurrency

Petri nets

- Merits:
  good at modeling and analyzing concurrent systems

- Demerits:
  ‘at’ model that is
  incomprehensible when system complexity increases

State oriented: Hierarchical concurrent FSM

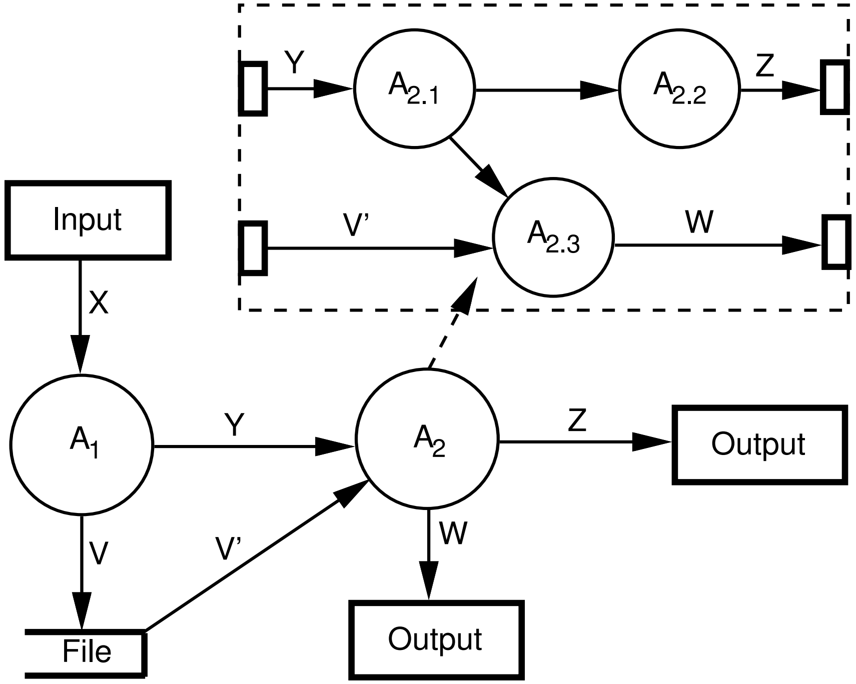


Hierarchical concurrent FSMs

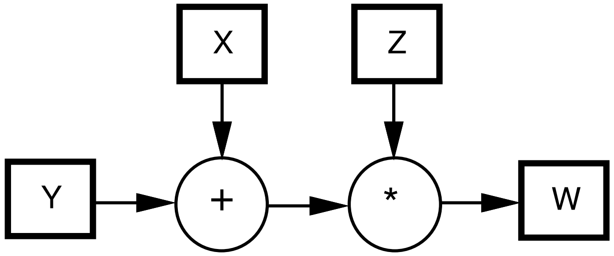
- Merits:
  support both hierarchy and concurrency
  good for representing complex systems

- Demerits:
  concentrate only on modeling control aspects
  and not data and activities

Activity oriented: Dataow graphs (DFG)



(a) Activity level

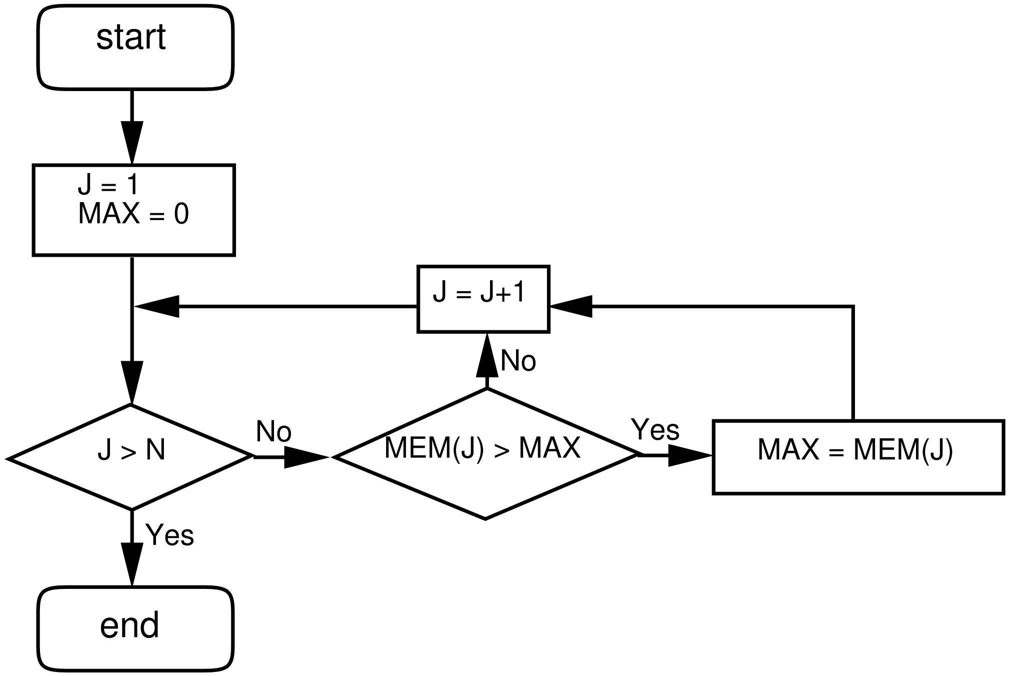


(b) Operation level

Dataow graphs

- Merits:
  - support hierarchy
  - suitable for specifying complex transformational systems
  - represent problem-inherent data dependencies
- Demerits:
  - do not express temporal behaviors or control sequencing
  - weak for modeling embedded systems

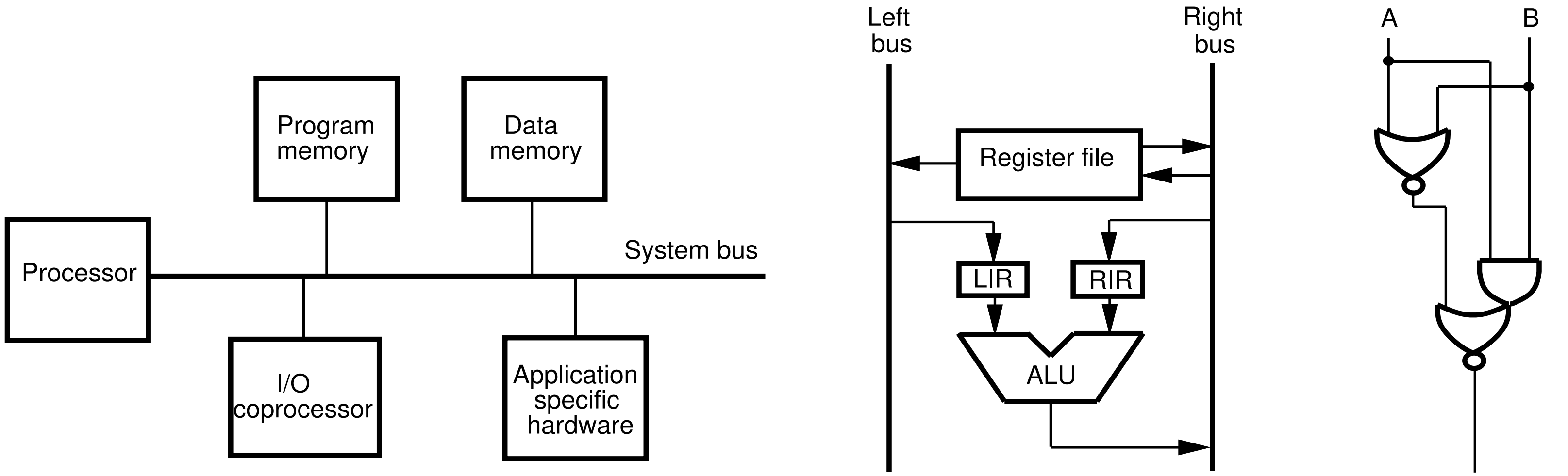
Activity oriented: Flowchart (CFG)



Flowcharts

- Merits:
  - useful to represent tasks governed by control ow
  - can impose a order to supersede natural data dependencies
- Characteristics:
  - used only when the system’s computation is well known

Structure oriented: Component-connectivity diagrams



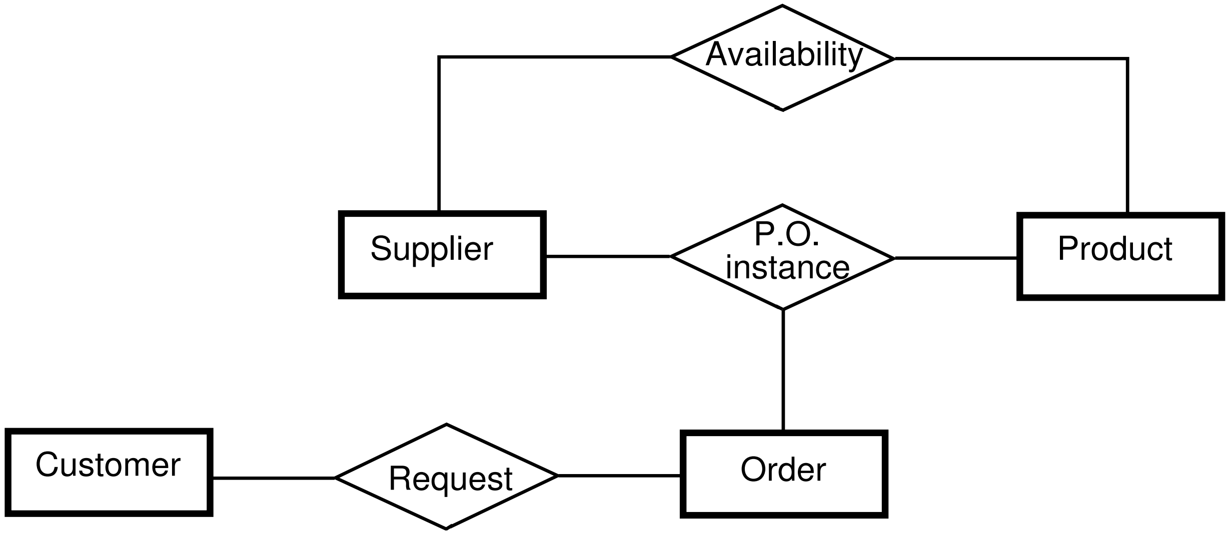
(a) Block diagram **(b) RT netlist** **(c) Gate netlist**

Component-connectivity diagrams

- Merits:
  good at representing system's structure

- Characteristics:
  often used in the later phases of design process

Data oriented: Entity-relationship diagram



Entity-relationship diagrams

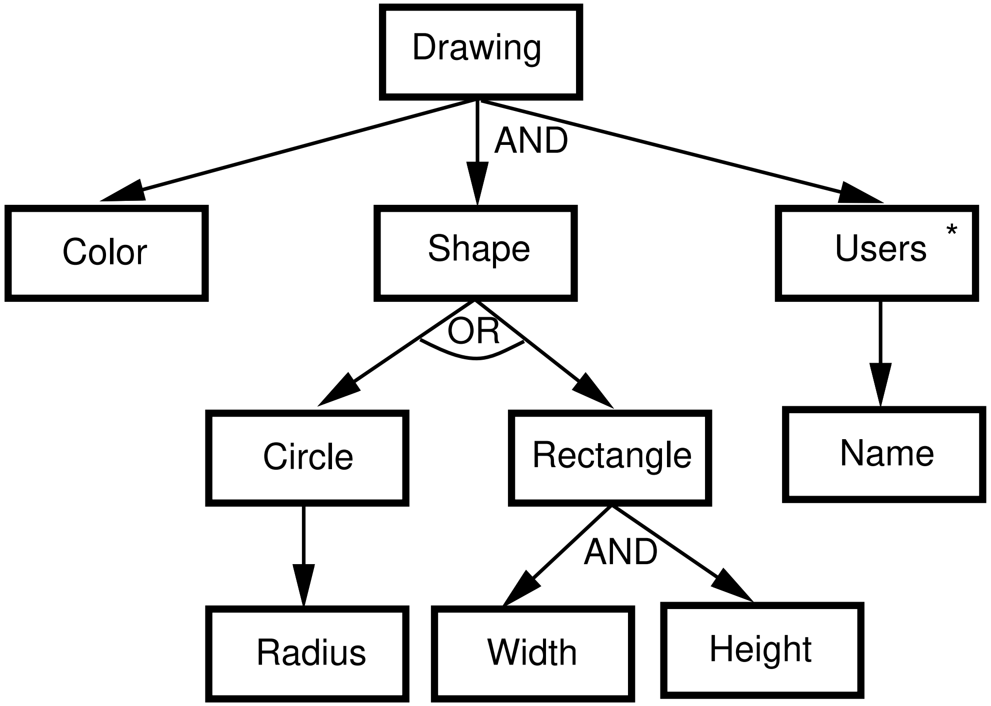
- Merits:

  provide a good view of the data in the system, also
  suitable for expressing complex relations among various kinds of data

- Demerits:

  do not describe any functional or temporal behavior of the system.

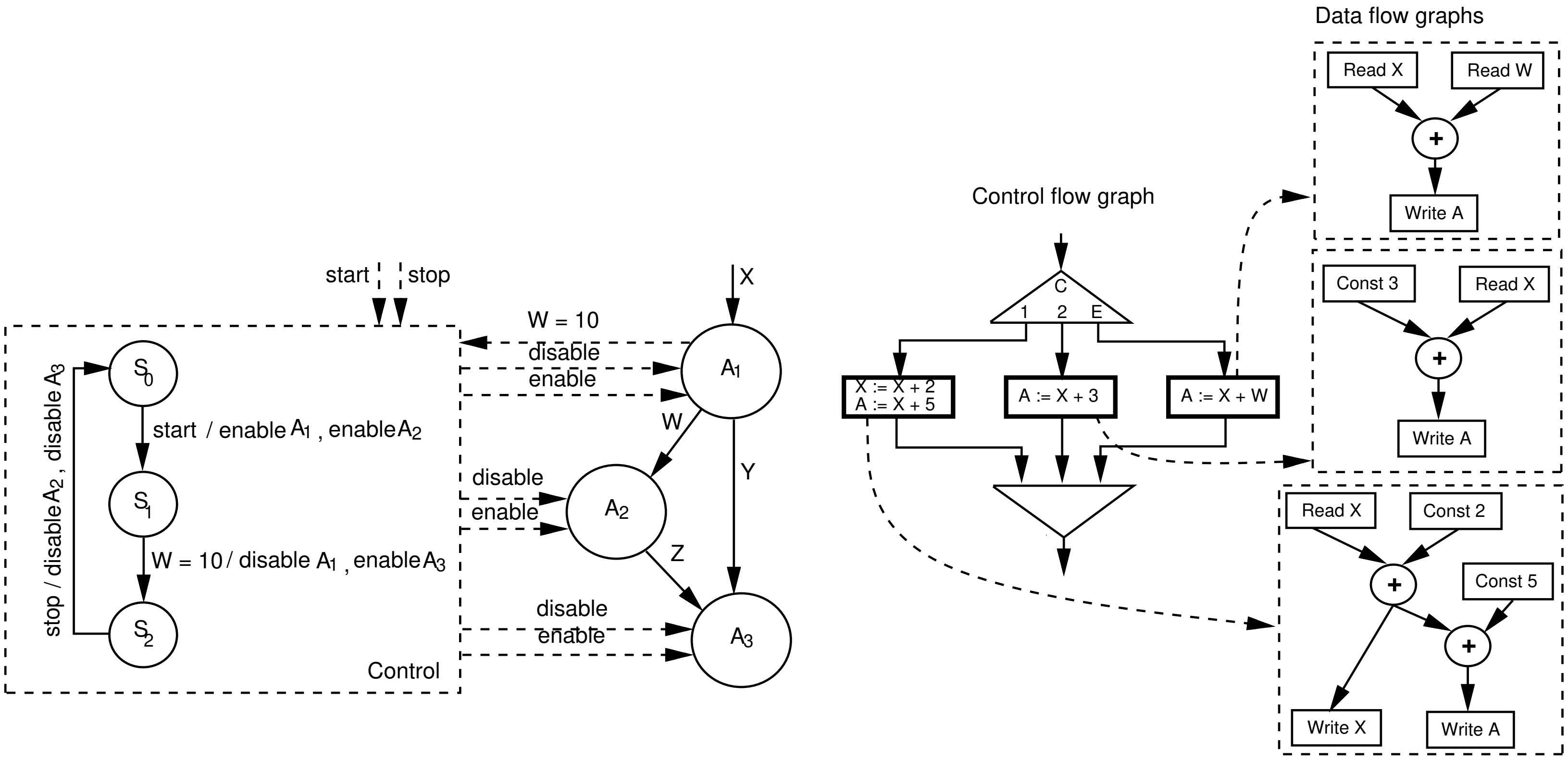
Data oriented: Jackson's diagram



Jackson's diagrams

- Merits:
  suitable for representing data having a complex composite structure.
- Demerits:
  do not describe any functional or temporal behavior of the system.

Heterogeneous: Control/dataow graph



(a) Activity level **(b) Operation level**

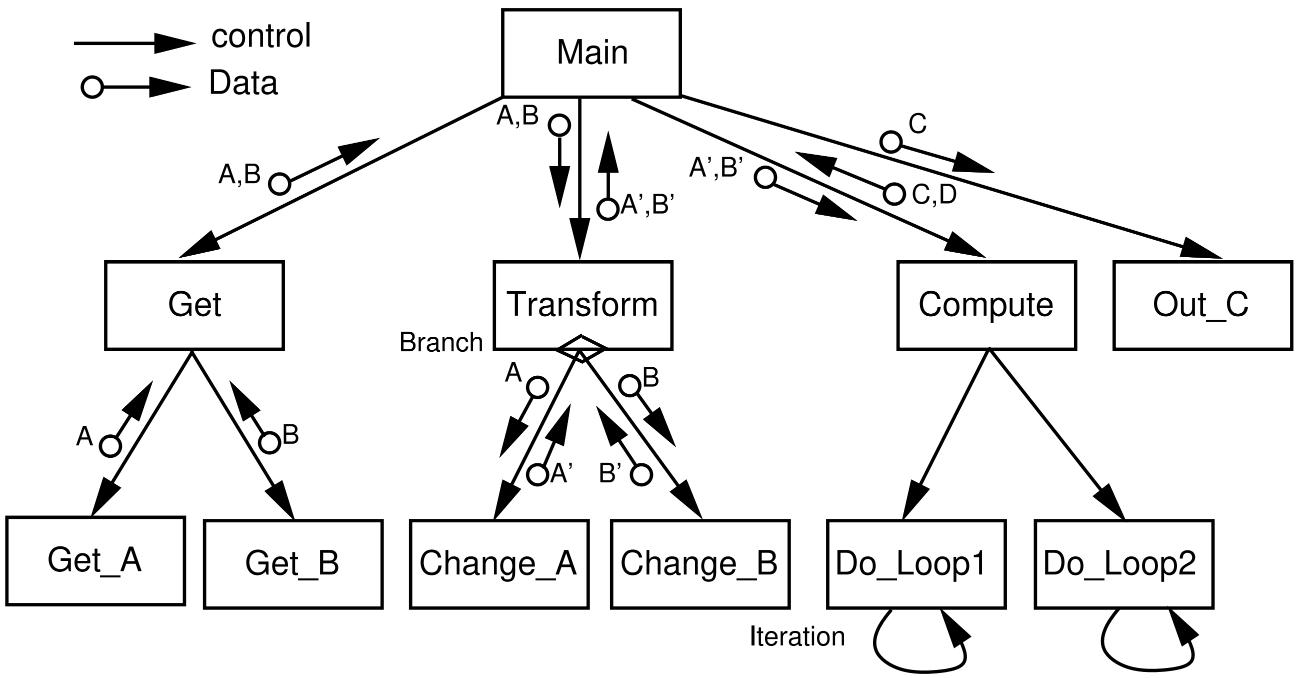
Control/dataow graphs

- Merits:

  correct the inability of DFG in representing the control of a system

  correct the inability of CFG to represent data dependencies

Heterogeneous: Structure chart



Structure charts

- Merits:
  represent both data and control

- Characteristics:
  used in the preliminary stages of program design

Heterogeneous: Programming languages

- Imperative vs declarative programming languages:
  C, Pascal, Ada, C++, etc.
  LISP, PROLOG, etc.

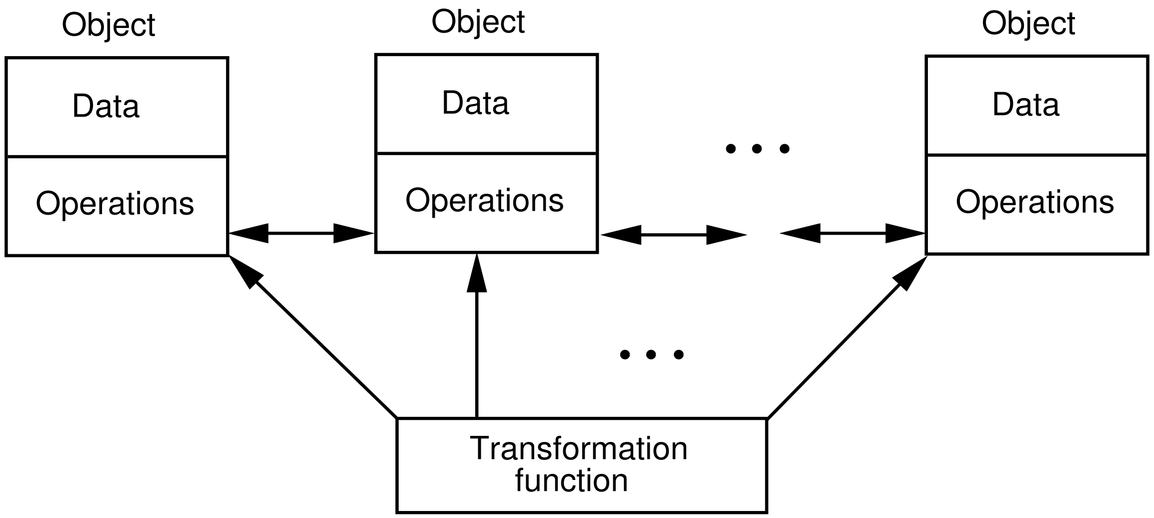
- Sequential vs concurrent programming languages:
  Pascal, C, etc.
  CSP, ADA, VHDL, etc.

Programming languages

- Merits:
  model data, activity, and control

- Demerits:
  do not explicitly model the system's states

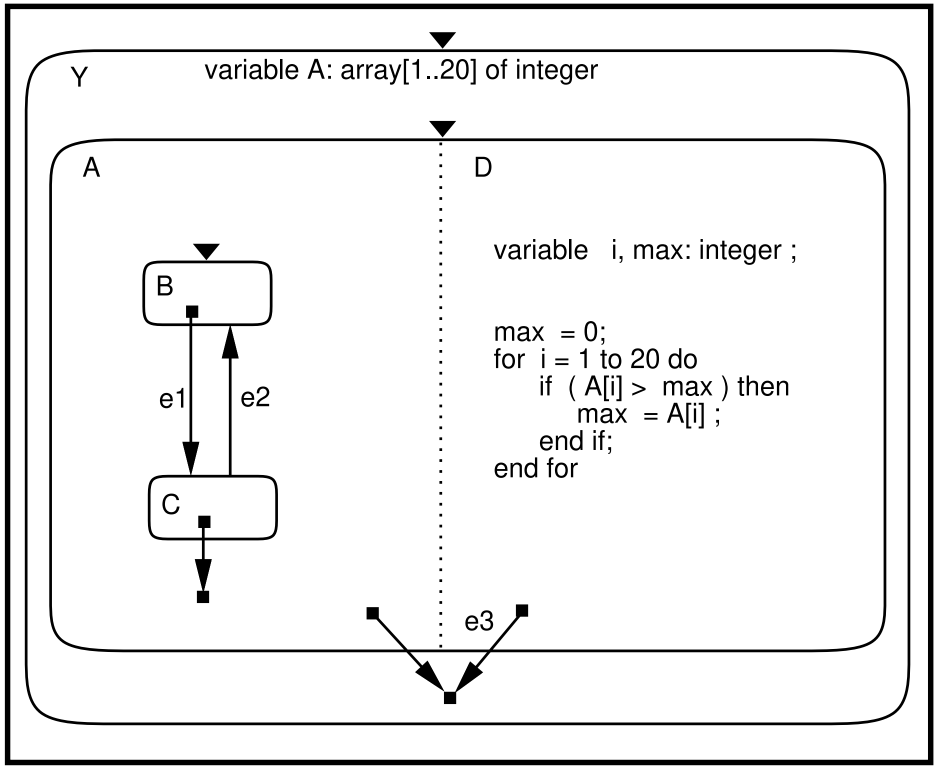
Heterogeneous: Object-oriented paradigm



Object-oriented paradigms

- Merits:
  support information hiding, inheritance, natural concurrency
- Demerits:
  not suitable for systems with complicated transformation functions

Heterogeneous: Program-state machine



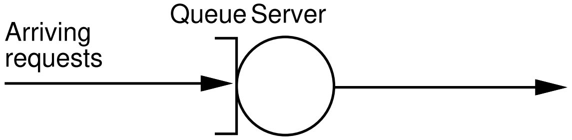
Program-state machines

- Merits:

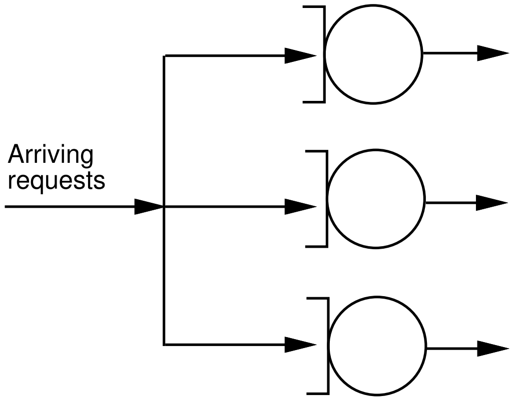
  represent system's states, data, control and activities in a single model

  overcome the limitations of programming languages and HCFSM models

Heterogeneous: Queueing model



(a) One server



(b) Multiple servers

Queueing model

- Characteristics:
  used for analyzing system's performance, and
  can nd utilization, queueing length, throughput

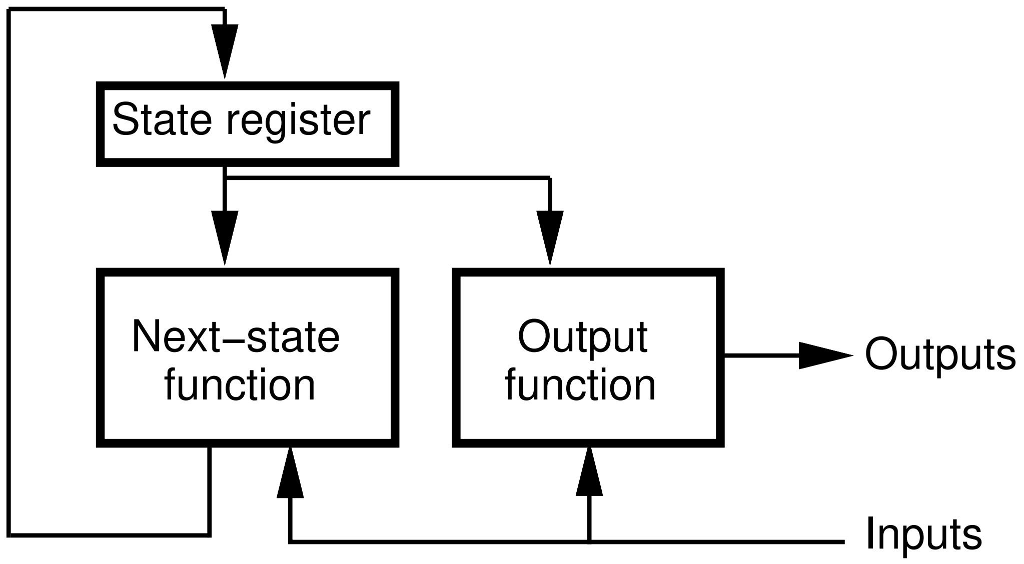
Architectures

- Application-specic architectures
  Controller architecture,
  Datapath architecture,
  Finite-state machine with datapath (FSMD).

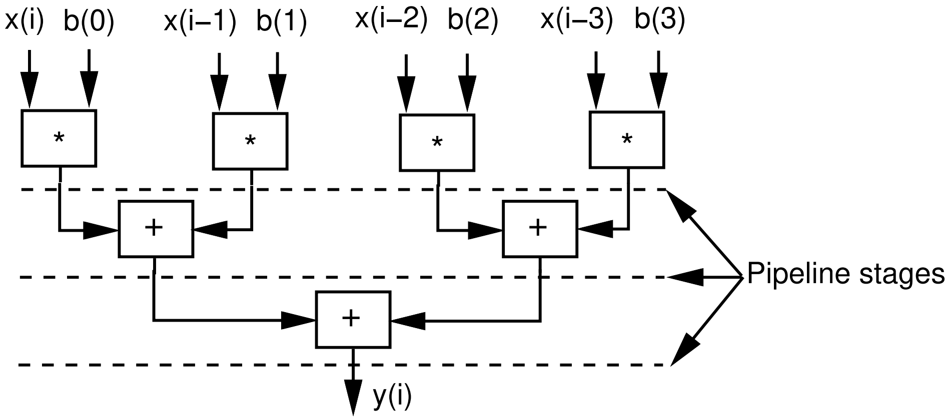
- General-purpose processors
  Complex instruction set computer (CISC)
  Reduced instruction set computer (RISC)
  Vector machine
  Very long instruction word computer (VLIW)

- Parallel processors

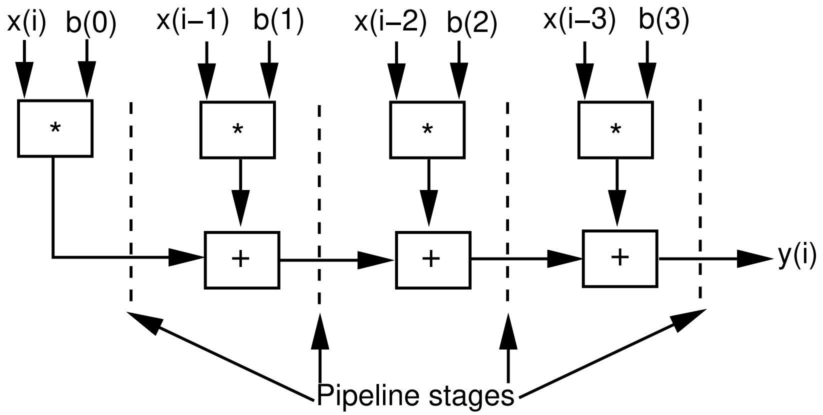
Controller architecture



Datapath architecture

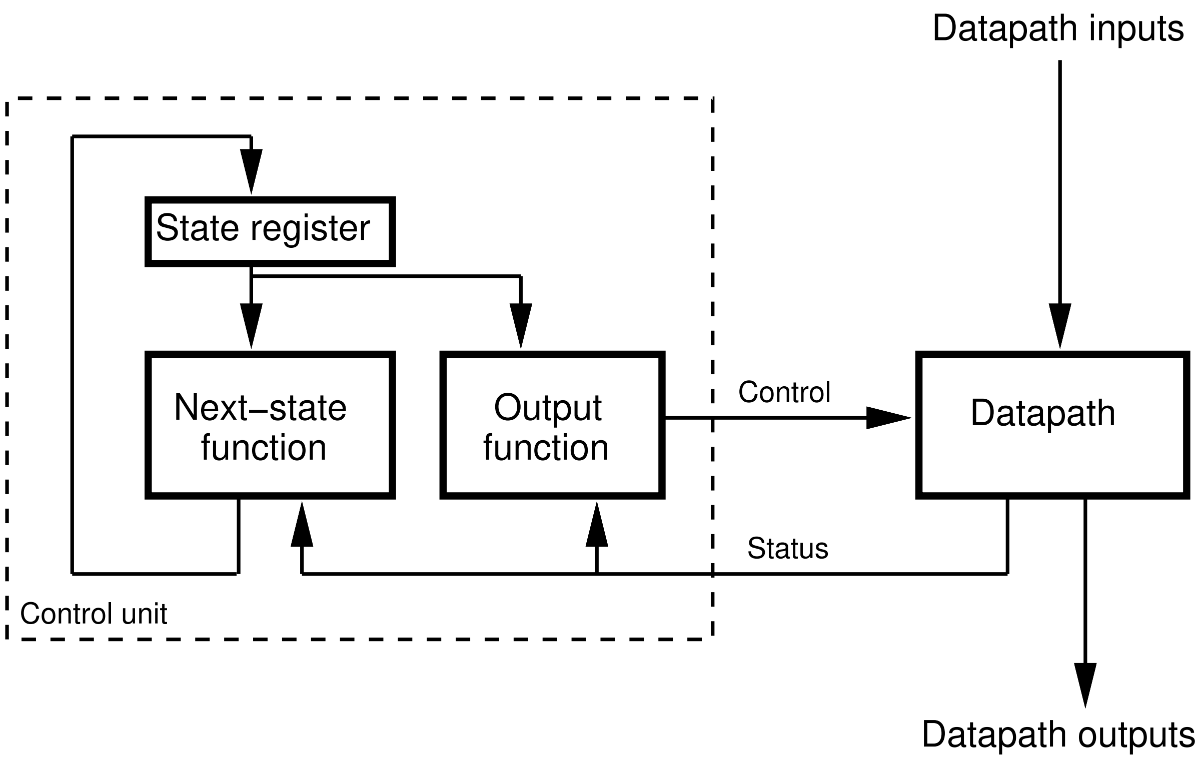


(a) Three stage pipeline



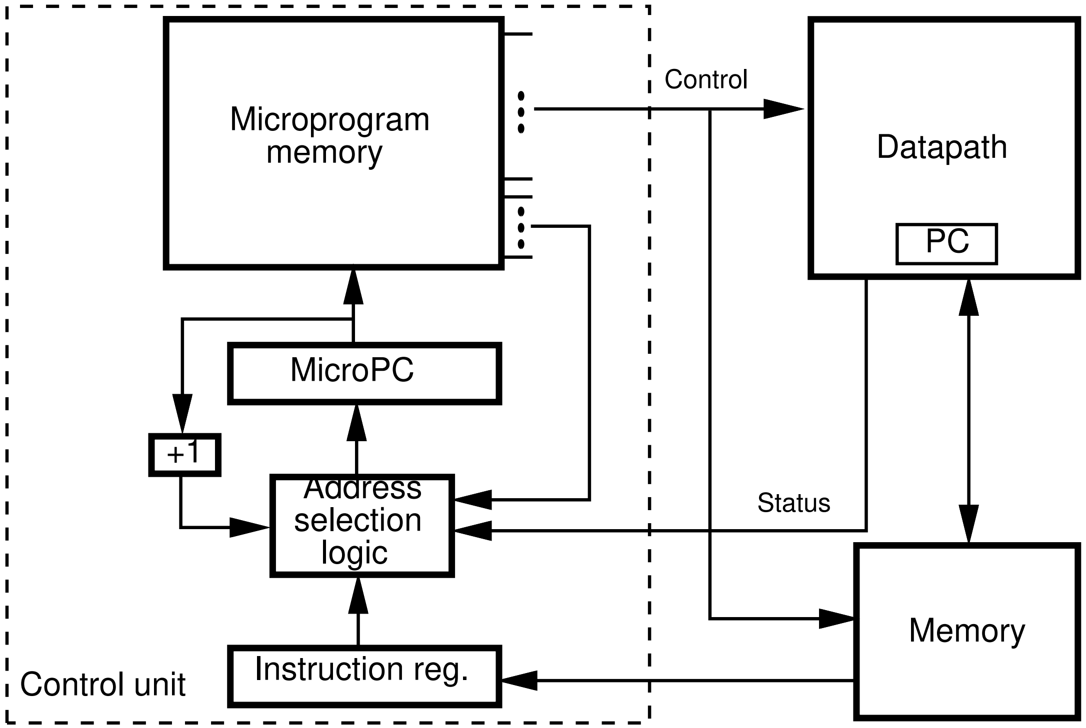
(b) Four stage pipeline

FSMD

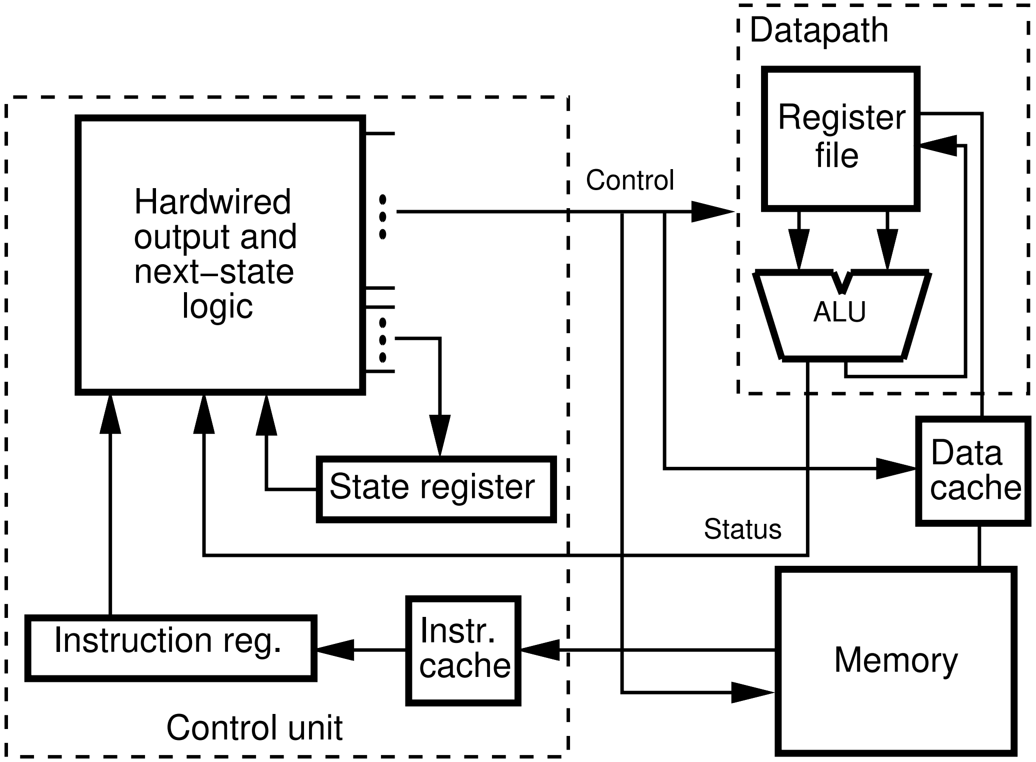


Datapath inputs
State register
Next–state function
Output function
Control
Datapath
Status
Control unit
Datapath outputs

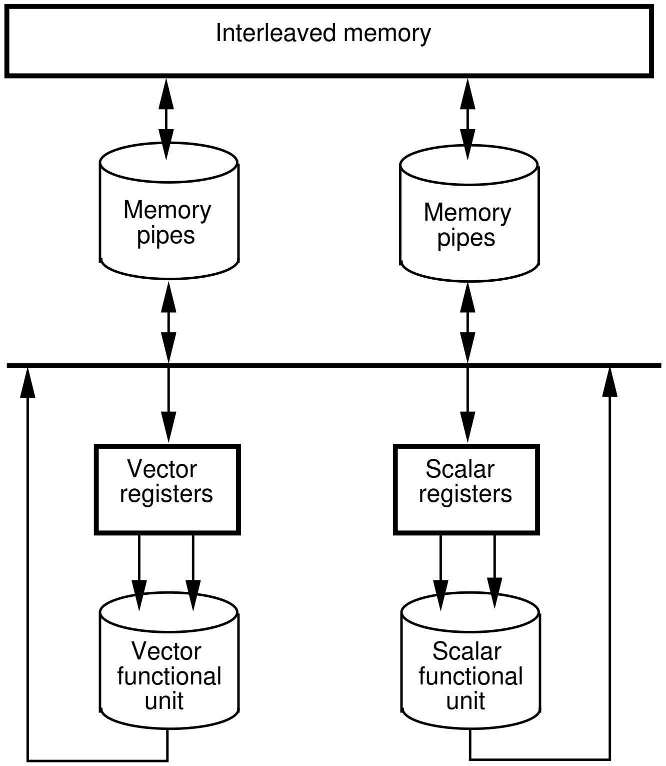
CISC architecture



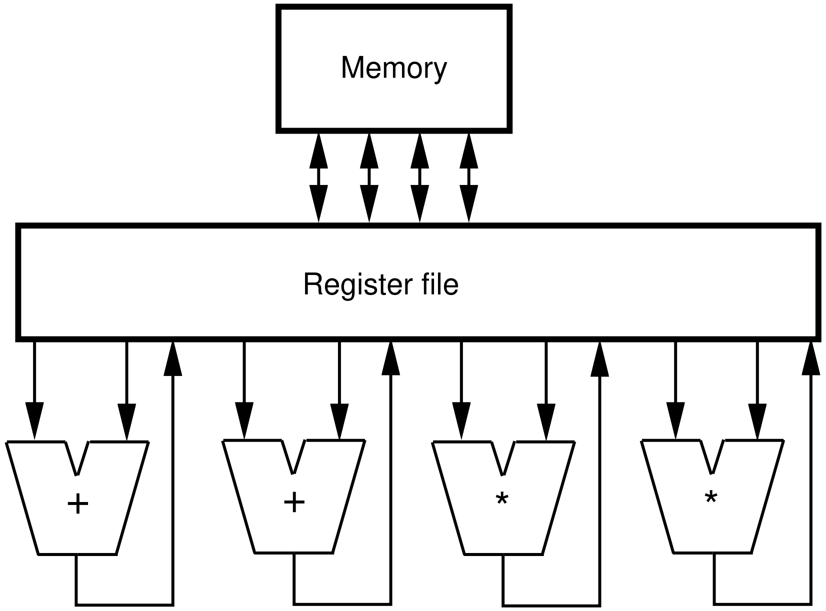
RISC architecture



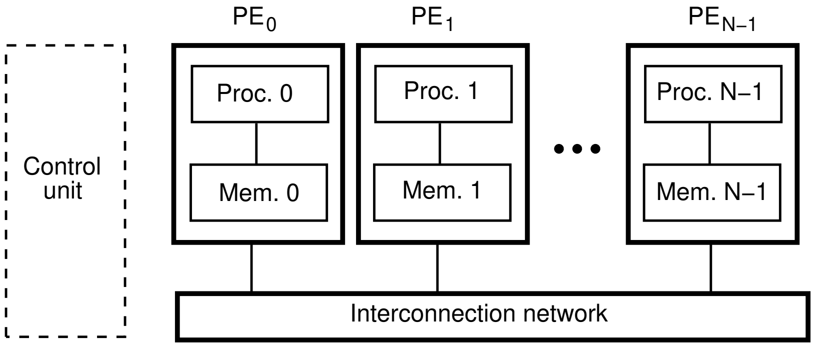
Vector machines



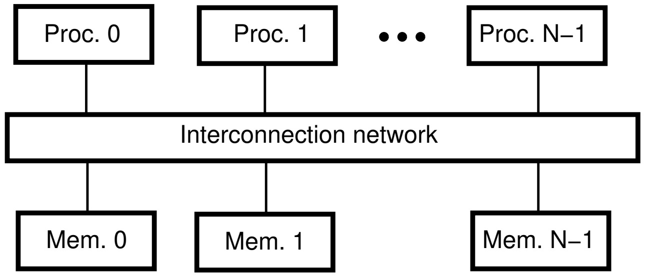
VLIW architecture



Parallel processors: SIMD/MIMD



(a) Message passing



(b) Shared memory

Conclusion

- Different models focus on different aspects
- Proper model needs to represent system's features
- Models are implemented in architectures
- Smooth transformation of models to architectures increases productivity

System specication

- For every design, there exists a **conceptual view**
- Conceptual view depends on application
  - Computation : conceptualized as a program
  - Controller : conceptualized as a state-machine
- **Goal** of specication language
  - Capture conceptual view with minimum designer effort
- Ideal language

  1-to-1 mapping between conceptual model & language constructs

Outline

- Characteristics of commonly used conceptual models:
  Concurrency, hierarchy, synchronization
- Requirements for embedded system specication
- Evaluate HDLs with respect to embedded systems
  VHDL, Verilog, Esterel, CSP, Statecharts, SDL, SpecCharts

Concurrency

- **Behavior:** a chunk of system functionality
  - e.g. process, procedure, state-machine

- System often conceptualized as set of concurrent behaviors

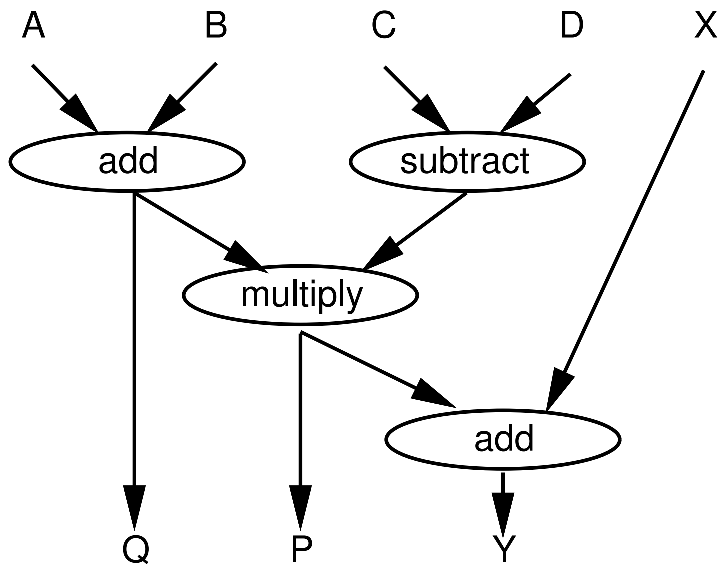
- Concurrency can exist at different abstraction levels:
  - Job-level
  - Task-level
  - Statement-level
  - Operation-level
  - Bit-level

- Two types of concurrency within a behavior
  - Data-driven, Control-driven

Data-driven concurrency

- Operations execute when input data is available
- Execution order determined by data dependencies

1: Q = A + B
2: Y = X + P
3: P = (C – D) * Q

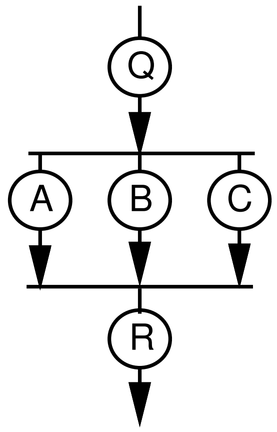


Control-driven concurrency

- Control thread : set of operations executed sequentially
- Concurrency represented by multiple control threads

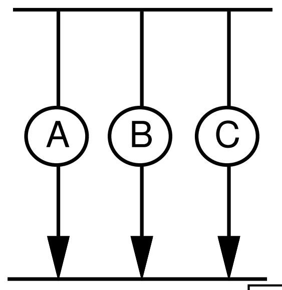
Fork-join statement

```
sequential  behavior X
begin
    Q();
    fork   A();  B(); C();  join;
    R();
end behavior X;
```



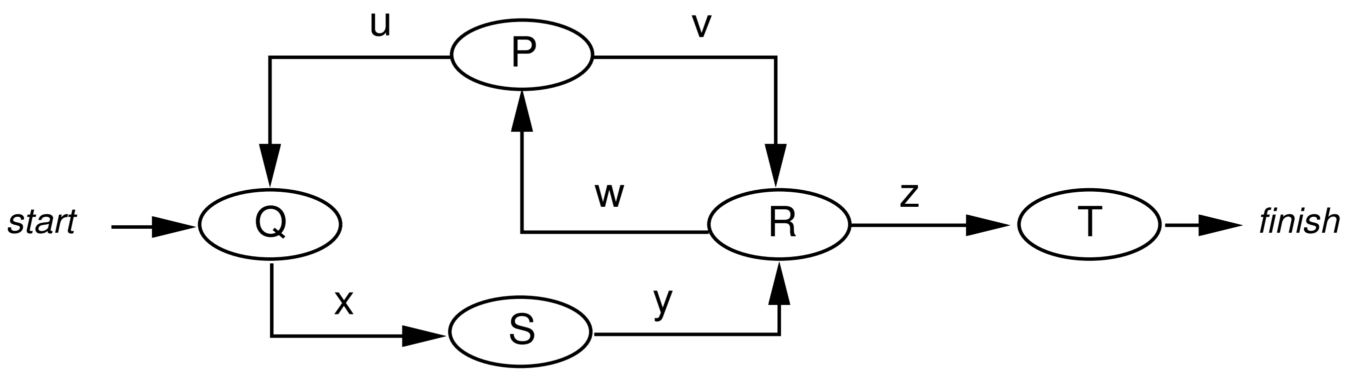
Process statement

```
concurrent  behavior X
begin
    process A();
    process B();
    process C();
end behavior X;
```



State-transitions

- Systems often are state-based, e.g. controllers
- State may represent
  - *mode* or stage of being
  - computation
- Difcult to capture using programming constructs

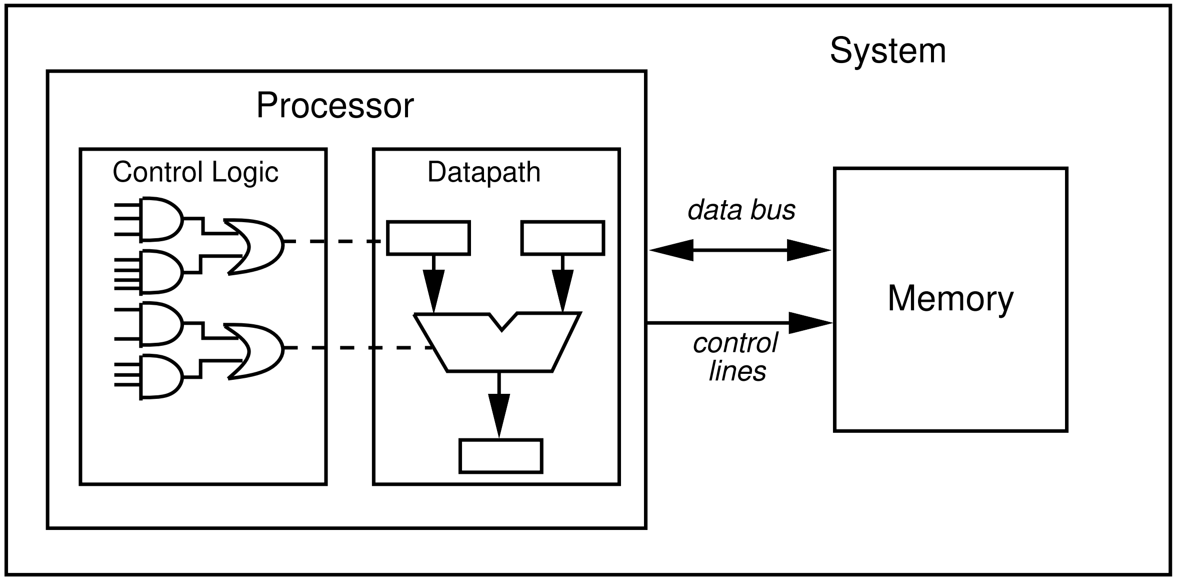


Hierarchy

- Required for managing system complexity
  - Allows system modeler to focus on one subsystem at a time
  - Enhances comprehension of system functionality
  - Scoping mechanism for objects like types and variables
- Two types of hierarchy
  - Structural hierarchy
  - Behavioral hierarchy

Structural hierarchy

- System represented as set of interconnected components
- Interconnections between components represent wires
- Several levels: systems, chips, RT-components, gates

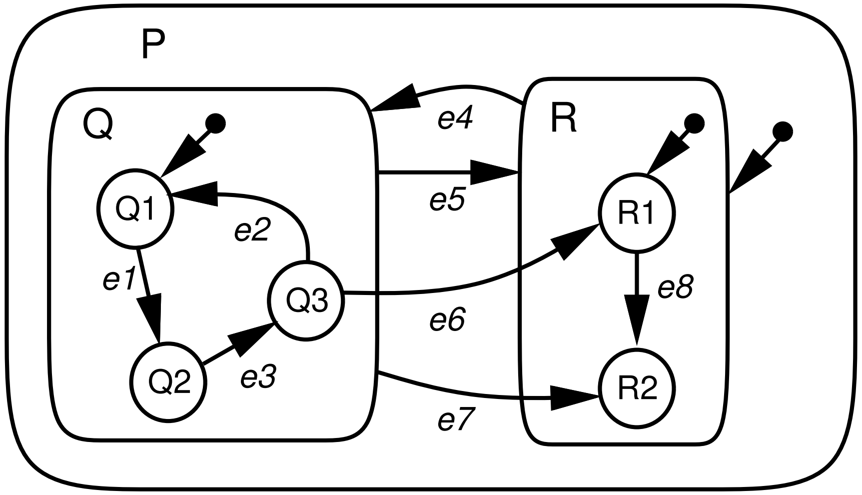


Behavioral hierarchy

- Ability to successively decompose behavior into sub-behaviors

```
behavior  P
    variable x, y;
begin
    Q(x) ;
    R(y) ;
end  behavior  P;
```

- Concurrent decomposition
  - Fork-join
  - Process
- Sequential decomposition
  - Procedure
  - State-machine



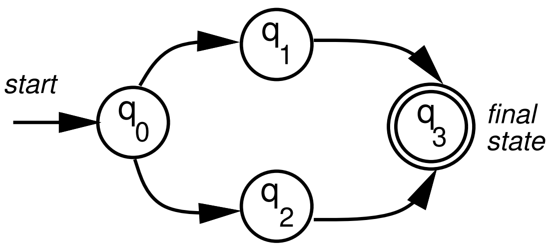
Programming constructs

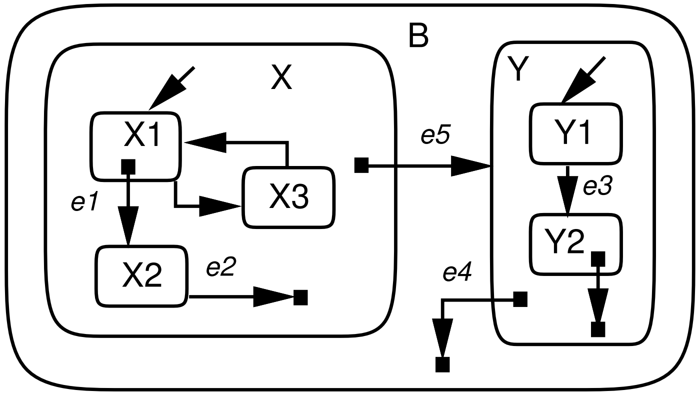
- Some behaviors easily conceptualized as sequential algorithms
- Wide variety of constructs available
  Assignment, branching, iteration, subprograms,
  recursion, complex data types (records, lists)

```
type      buffer_type  is array (1 to 10) of integer;
variable  buf : buffer_type;
variable  i, j : integer;

for i = 1 to 10
    for j = i to i
        if (buf(i) > buf(j)) then
            SWAP(buf(i), buf(j));
        end if;
    end for;
end for;
```

Behavioral completion

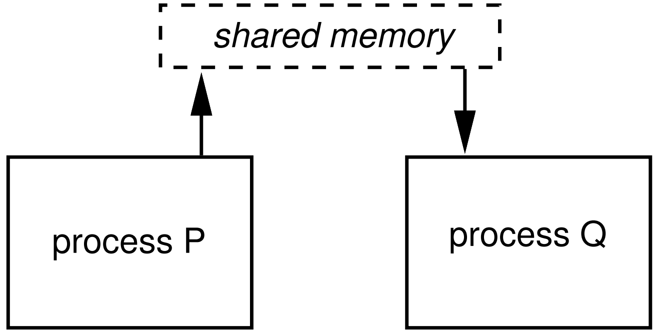
- Behavior *completes* when all computations performed
- Advantages
  - Behavior can be viewed without inter-level transitions
  - Allows natural decomposition into sequential subbehaviors





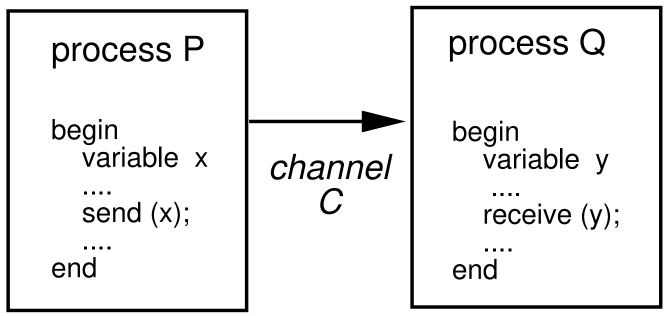
Communication

- Concurrent behaviors exchange data



- Shared-memory model
  - Sender updates common medium
  - Persistent, Non-persistent

- Message-passing model
  - Data sent over abstract *channels*
  - Unidirectional / bidirectional
  - Point-to-point / multiway
  - Blocking / non-blocking



Synchronization

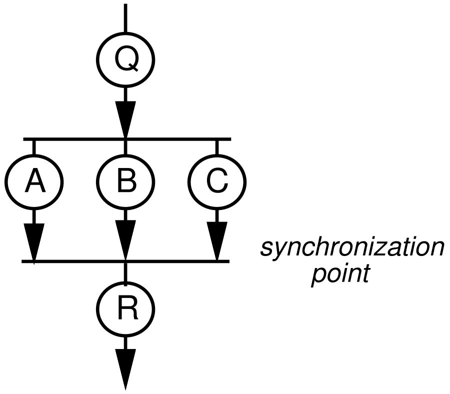
- Concurrent behaviors execute at different speeds
- Synchronization required when
  - Data exchanged between behaviors
  - Different activities must be performed simultaneously
- Two types of synchronization mechanisms
  - Control-dependent
  - Data-dependent

Control-dependent synchronization

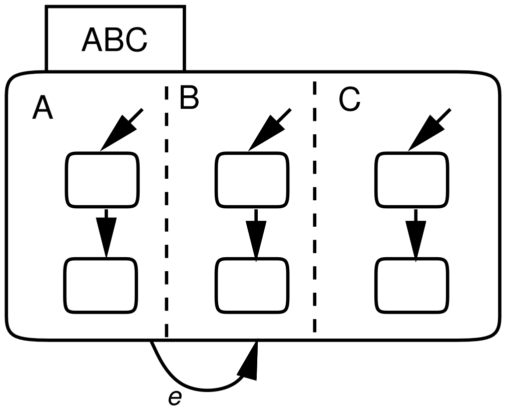
- Synchronization based on control structure of behavior

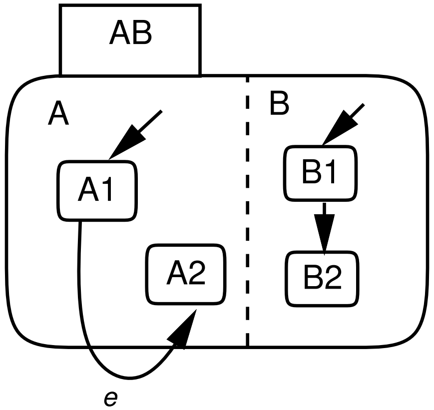
Fork-join

```
behavior X
begin
    Q();
    fork   A();  B(); C();  join;
    R();
end behavior X;
```



Reset

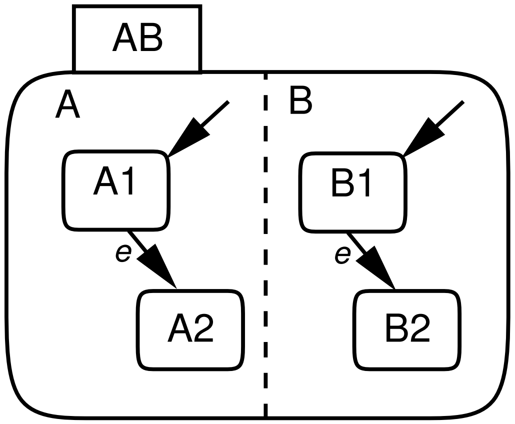




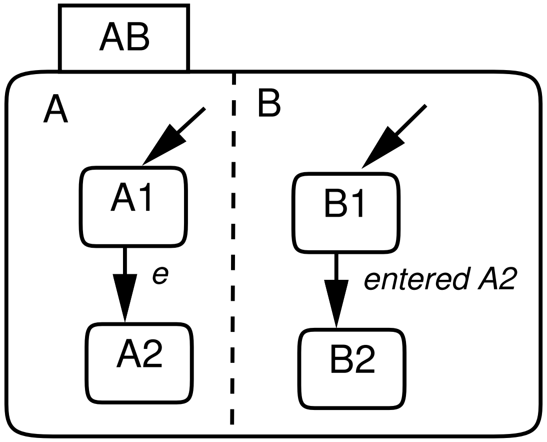
UC Irvine

Data-dependent synchronization

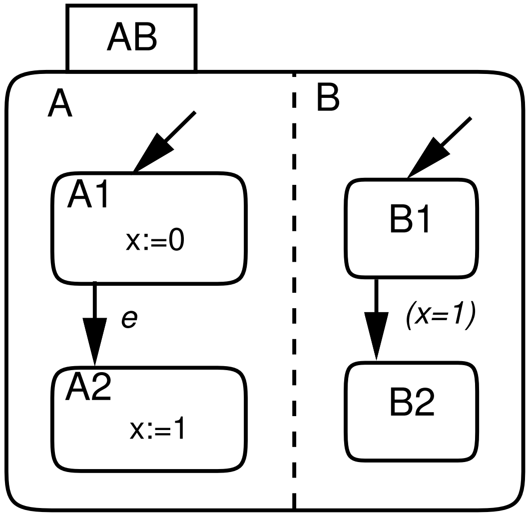
- Synchronization based on communication of data between behaviors



Synchronization by common event



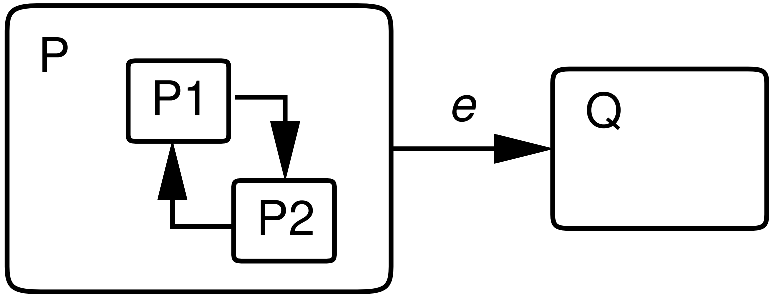
Synchronization by status detection



Synchronization by common variable

Exception handling

- Occurrence of event terminates current computation
- Control transferred to appropriate next mode
- Example of exceptions: interrupts, resets

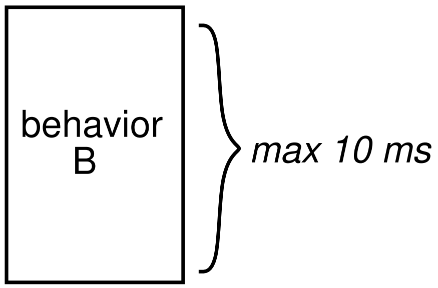


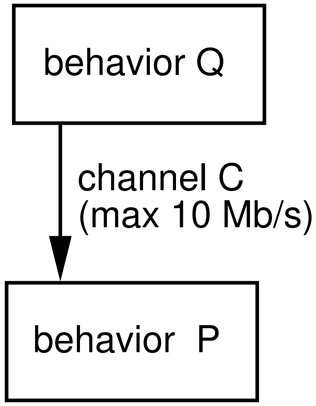
Timing

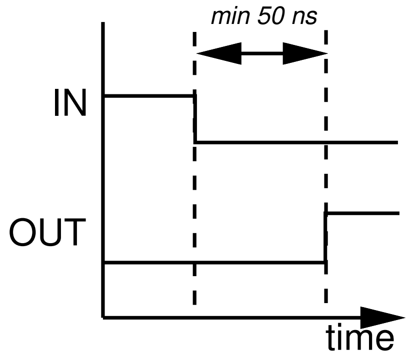
- Required to represent real world implementations
- **Functional timing**: affects simulation of system specication

  wait for 200 ns;

  A <= A + 1 after 100 ns;
- **Timing constraints**: guide synthesis and verication tools



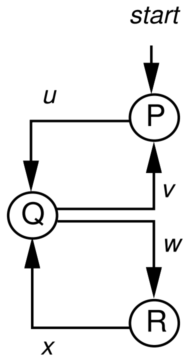


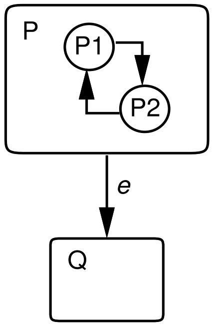


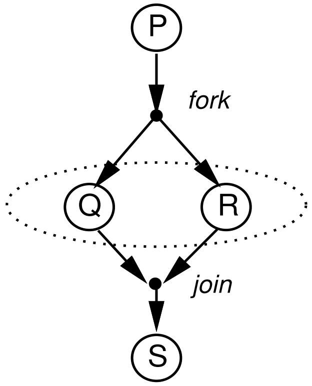
Embedded system specication

- Embedded system: behavior dened by interaction with environment
- Essential characteristics

State-transitions	Exceptions
Behavioral hierarchy	Concurrency
Programming constructs	Behavioral completion







VHDL

- IEEE standard, intended for documentation and exchange of designs [IEE88]

- **Characteristics supported**
  - Behavioral hierarchy : single level of processes
  - Structural hierarchy : nested blocks and component instantiations
  - Concurrency : task-level (process), statement-level (signal assignment)
  - Programming constructs
  - Communication : shared-memory using global signals
  - Synchronization : *wait on* and *wait until* statements
  - Timing : *wait for* statement, *after* clause in assignments

- **Characteristics not supported**
  - Exceptions : partially supported by guarded signal assignments
  - State transitions

Verilog and Esterel

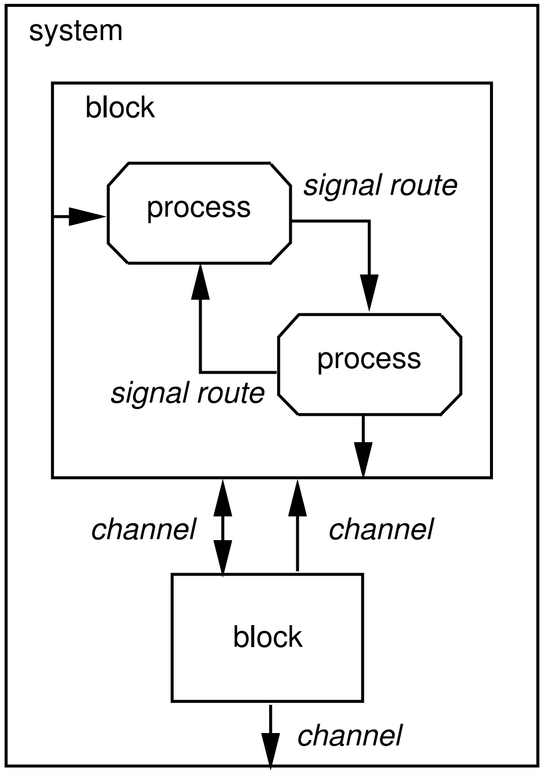
- Verilog [TM91] developed as proprietary language for specication, simulation
- Esterel [Hal93] developed for specication of reactive systems
- **Characteristics supported:**
  - Behavioral hierarchy : fork-join
  - Structural hierarchy : hierarchy of interconnected *modules*
  - Programming constructs
  - Communication : shared registers (Verilog) and broadcasting (Esterel)
  - Synchronization : *wait* for an event on a signal
  - Timing : modeling of gate, net, assignment delays in Verilog
  - Exceptions : *disable* (Verilog), *watching, do-upto, trap* statements (Esterel)
- **Characteristics not supported:** State transitions

SDL (Specication and Description language)

- CCITT standard in telecommunication for protocol specication [BHS91]

- **Characteristics supported**
  - Behavioral hierarchy : nested dataow
  - Structural hierarchy : nested blocks
  - State transitions : state machine in processes
  - Communication : message passing
  - Timing : *timeouts* generated by timer object

- **Characteristics not supported**
  - Exceptions
  - Programming constructs



CSP (Communicating Sequential Processes)

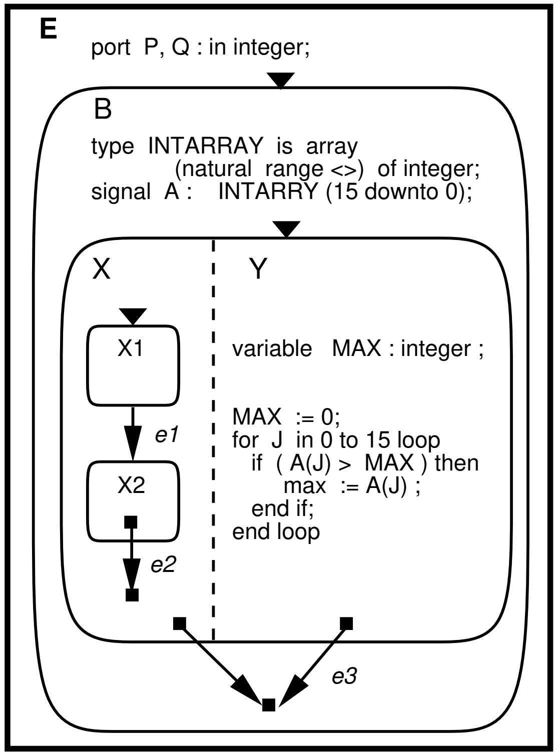
- Intended to specify programs running on multiprocessor machines [Hoa78]

- **Characteristics supported**
  - Behavioral hierarchy : fork-join using *parallel* command
  - Programming constructs
  - Communication : message passing using *input, output* commands
  - Synchronization : blocking message passing

- **Characteristics not supported**
  - Exceptions
  - State transitions
  - Structural hierarchy
  - Timing

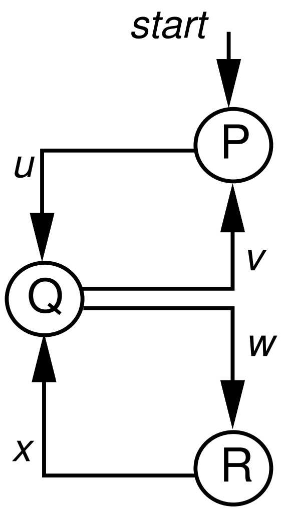
SpecCharts

- Developed for embedded system specication [NVG92]
- PSM (program-state machine) model + VHDL
- **Characteristics supported**
  - Behavioral hierarchy : sequential/concurrent behaviors
  - State transitions: TOC (transition on completion) arcs
  - Communication : shared memory, message passing
  - Exceptions : TI (transition immediately) arcs
- **Characteristics similar to VHDL**
  - Programming constructs
  - Structural hierarchy
  - Synchronization and Timing



SpecCharts : state transitions

- State transitions represented by TOC and TI arcs between behaviors



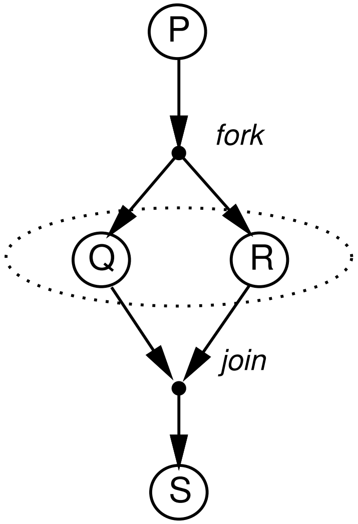
behavior MAIN type sequential subbehaviors is
begin
    P : (TOC, u, Q) ;
    Q : (TOC, v, P), (TOC, w, R);
    R : (TOC, x, Q);

    behavior P .....
    behavior Q .....
    behavior R .....

end MAIN;

SpecCharts : behavioral hierarchy

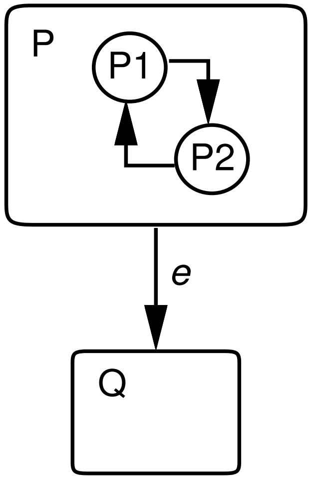
- Hierarchy represented by nested behaviors
- Behavior decomposed into sequential or concurrent subbehaviors




```
behavior MAIN  type sequential subbehaviors is
begin
    P :  (TOC, true, Q_R);
    Q_R  :  (TOC, true, S);
    S : ;

    behavior  P .....

    behavior  Q_R  type concurrent subbehavior is
    begin
        Q : (TOC, true, halt);
        R : (TOC, true, halt);

        behavior Q .....
        behavior R .....
    end Q_R;

    behavior  S.....
end MAIN;
```

SpecCharts : exceptions

- Exceptions represented by TI (transition immediately) arcs




```
behavior MAIN  type sequential subbehaviors is
begin
    P  :  (TI, e, Q);
    Q  :  ;

    behavior  P
        behavior P1
            .......
        behavior P2
            .......

    behavior  Q
        ......

end MAIN;
```

Summary

Language	Embedded System Features					
	State Transitions	Behavioral Hierarchy	Concurrency	Program Constructs	Exceptions	Behavioral Completion
VHDL	○	◐	●	●	○	●
Verilog	○	●	●	●	●	●
Esterel	○	●	●	●	●	●
SDL	●	◐	●	○	○	●
CSP	○	●	●	●	○	●
Statecharts	●	●	●	○	●	○
SpecCharts	●	●	●	●	●	●

● Feature fully supported ◐ Feature partially supported ○ Feature not supported

Specication example

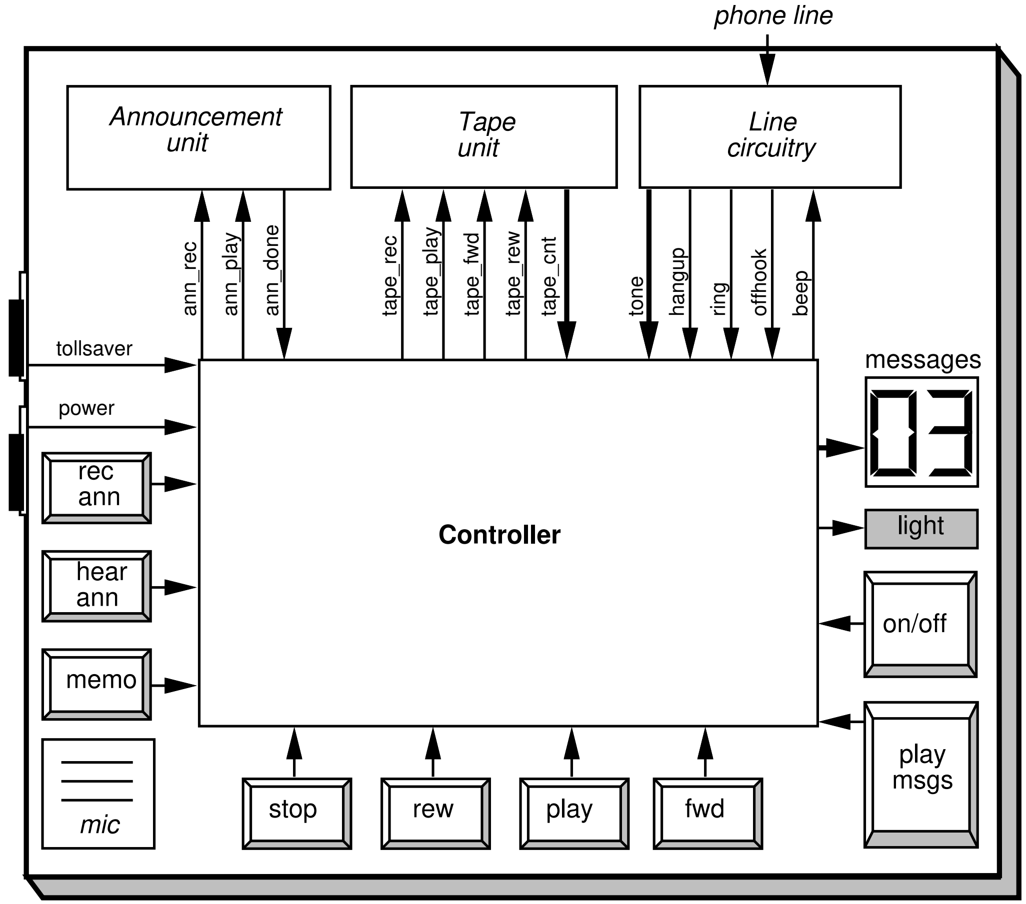
- An executable specication-language enables:
  - Early verication
  - Precision
  - Automation
  - Documentation

- A good language/model match reduces:
  - Capture time
  - Comprehension time
  - Functional errors

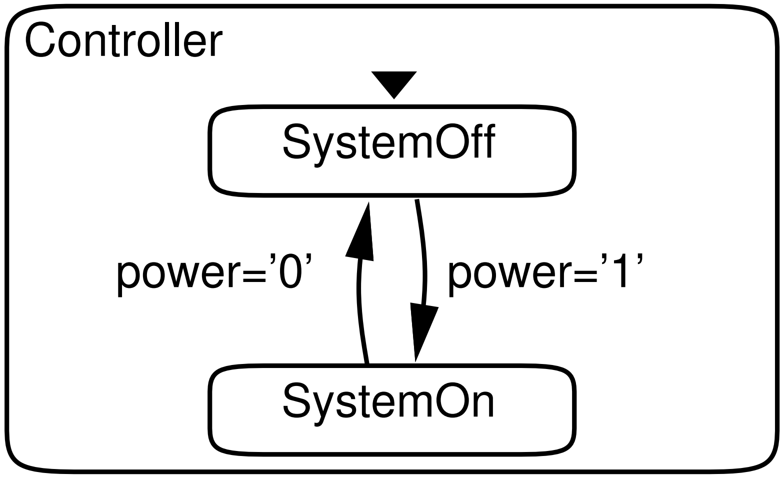
Outline

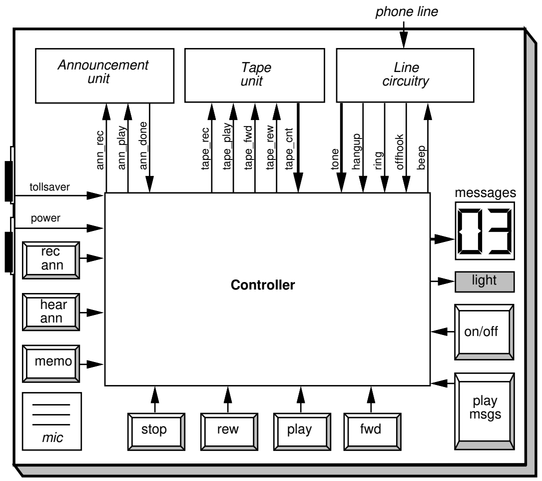
- Capture an example's model in a particular language
  - PSM model in the SpecCharts language
- Point out the benets of a good language/model match
- Highlight experiments that demonstrate those benets

Answering machine controller's environment



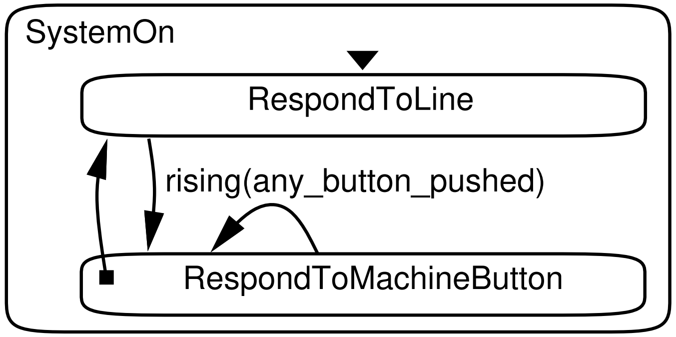
Highest-level view of the controller

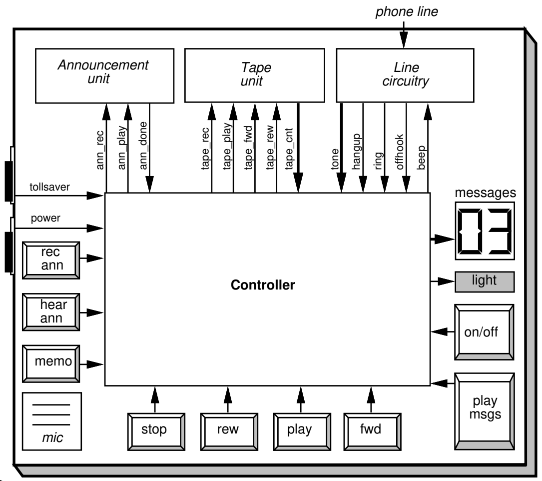




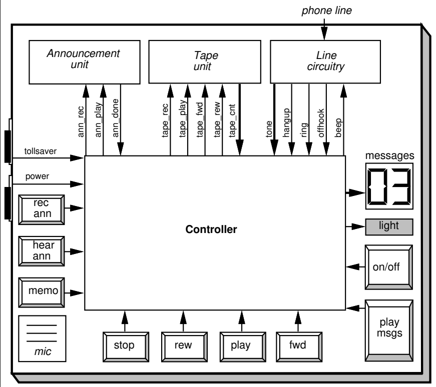
The SystemOn behavior

- System usually responds to the line
- Pressing any machine button gets immediate response



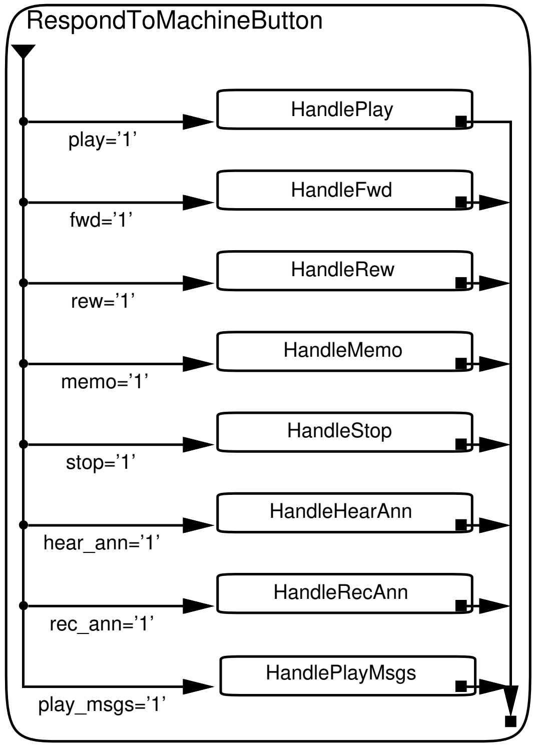


The RespondToMachineButton behavior




```
behavior  RespondToMachineButton
   type code is
begin
    if (play='1') then
        HandlePlay;
    elsif (fwd='1') then
        HandleFwd;
    elsif (rew='1') then
        HandleRew;
    elsif (memo='1') then
        HandleMemo;
    elsif (stop='1') then
        HandleStop;
    elsif (hear_ann='1')  then
        HandleHearAnn;
    elsif (rec_ann='1') then
        HandleRecAnn;
    elsif (play_msgs='1') then
        HandlePlayMsgs;
    end if;
end;
```

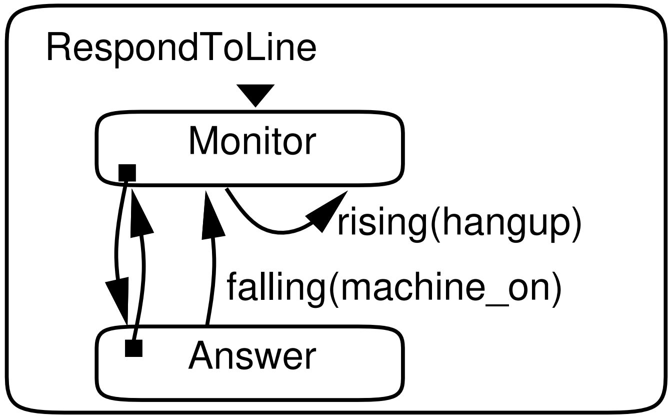
(a)

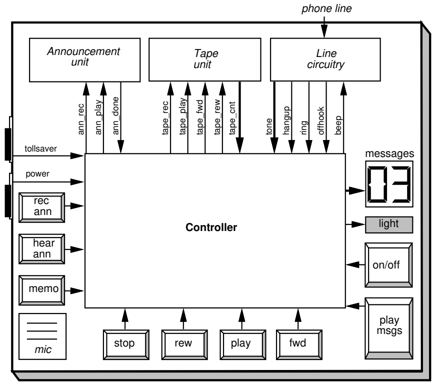


(b)

The RespondToLine behavior

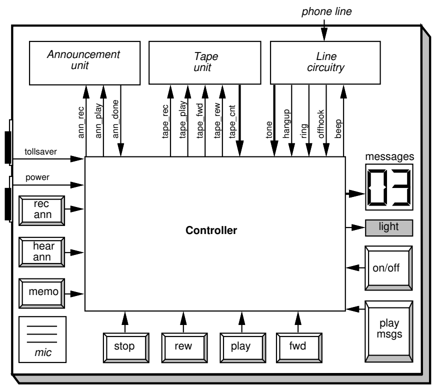
- Monitors line for rings
- Answers line
- Responds to exceptions
  - Hangup
  - Machine turned off





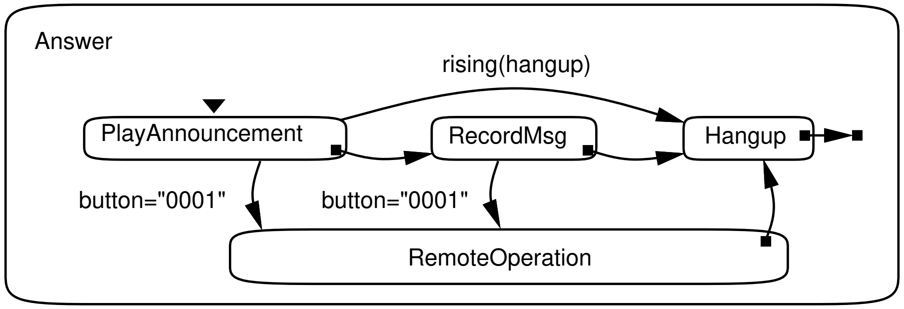
The Monitor behavior

- Counts for required rings
- Requirements may change




```
Monitor
        signal  rings_to_wait : integer range 1 to 20 := 4;
        function  DetermineRingsToWait return integer is begin
            if ((num_msgs > 0) and (tollsaver='1')  and (machine_on='1')) then
               return(2);
           elsif (machine_on='1') then
               return(4);
           else
               return(15);
           end if;
        end;

  MaintainRingsToWait                            | CountRings
                                                 |     variable I : integer range 0 to 20;
  loop                                           |
    rings_to_wait <= DetermineRingsToWait;       |     i := 0;
    wait on tollsaver, machine_on;               |     while (i < rings_to_wait) loop
  end loop;                                      |        wait on rings_to_wait, ring;
                                                 |        if (rising(ring)) then
                                                 |           i := i + 1;
                                                 |        end if;
                                                 |     end loop;
```

The Answer behavior



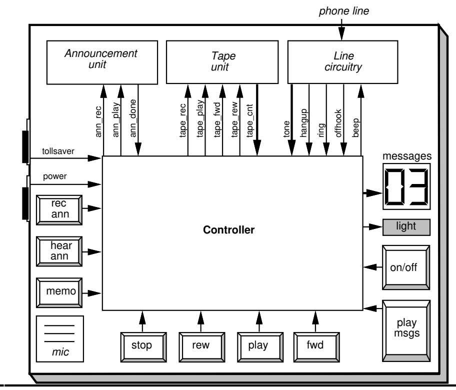
(a)

```
behavior  PlayAnnouncement  type code is
begin
   ann_play <= '1';
   wait until ann_done = '1';
   ann_play <= '0';
end;
```

(b)

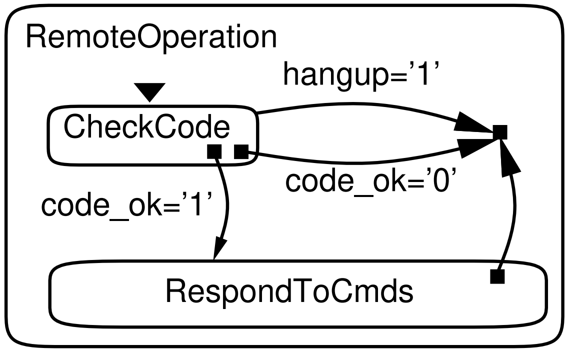
```
behavior  RecordMsg  type code is
begin
   ProduceBeep(1 s);
   if  (hangup = '0') then
      tape_rec <= '1';
      wait until hangup='1'  for 100 s;
      ProduceBeep(1 s);
      num_msgs <= num_msgs + 1;
      tape_rec <= '0';
   end if;
end;
```

(c)



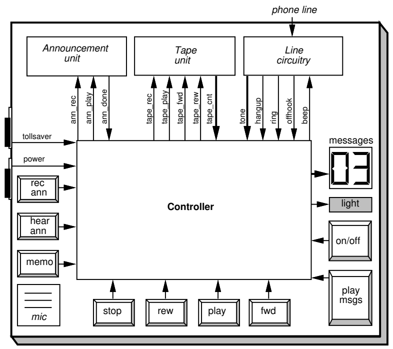
The RemoteOperation behavior

- Owner can operate machine remotely by phone
- Owner identies himself by four button ID



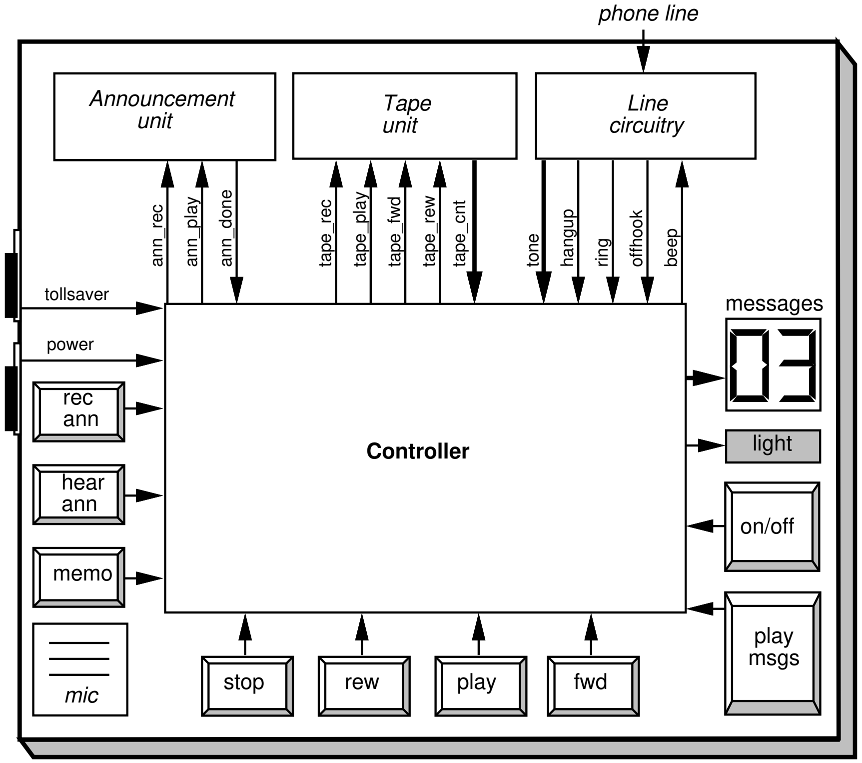

```
behavior  CheckUserCode  type code is
begin
   code_ok <= true;
   for (i in 1 to 4) loop
      wait until tone /= "1111" and tone'event;
      if (tone /= user_code(i)) then
          code_ok <= false;
      end if;
   end loop;
end;
```

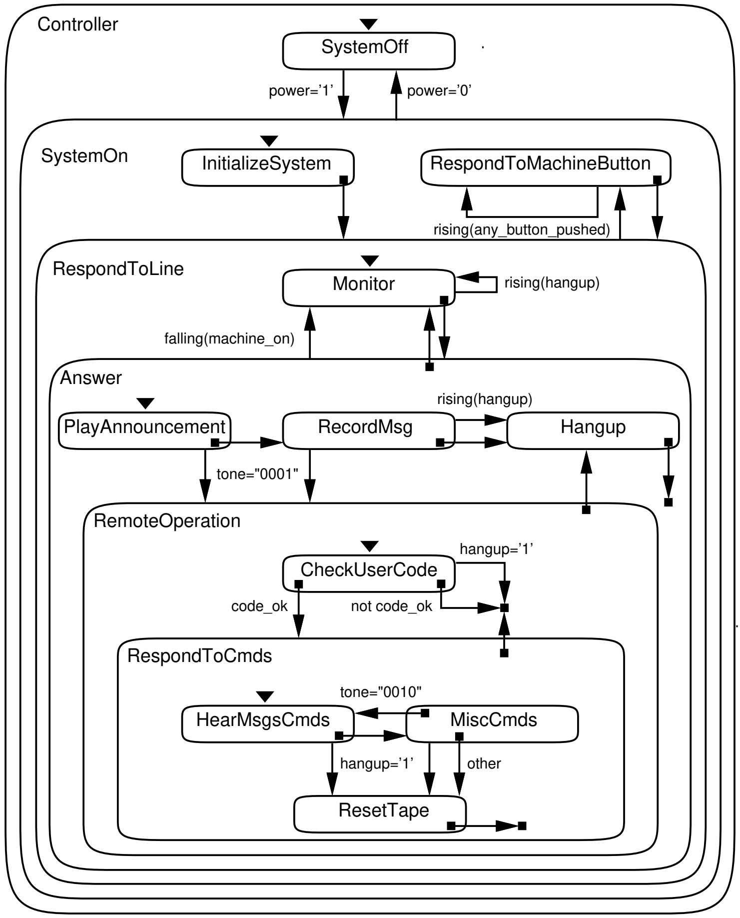
(a) **(b)**



UC Irvine

The answering machine controller specication





UC Irvine

Executable specication use

- Precision
  - Readability/precision compete in a natural language
  - Executable specication encourages precision
  - Designer asks questions, specication answers them
- Language/model match (SpecCharts/PSM):
  - Hierarchy
  - State-transitions
  - Programming constructs
  - Concurrency
  - Exceptions
  - Completion
  - Equivalence of states and programs

Specication capture experiment

	VHDL	SpecCharts
Average specification–time in minutes	40	16
Number of modelers	3	3
Number of incorrect specifications first time	2	0
Number of incorrect specifications second time	1	0

- VHDL modelers required 2.5 times longer
- Two VHDL specications possessed control errors
- SpecCharts were effective for state-transitions and exceptions

Comparison of SpecCharts, VHDL and Statecharts

Answering machine example

		Conceptual model	SpecCharts	VHDL (hierarch.)	VHDL (flat)	Statecharts
Specification attributes	Program−states	42	42	42	32	80
	Arcs	40	40	40	152	135
	Control signals	−−	0	84	1	0
	Lines/leaf	−−	7	27	29	−−
	Lines	−−	446	1592	963	−−
	Words	−−	1733	6740	8088	−−
Shortcomings	No sequential program constructs					X
	No hierarchy			X	X	
	No exception constructs			X	X	
	No hierarchical events				X	
	No state−transition constructs			X	X	

Design quality experiment

Design attribute	Designed from English	Designed from SpecCharts
Control transistors	3130	2630
Datapath transistors	2277	2251
Total transistors	5407	4881
Total pins	38	38

- No loss in design quality with an executable language

Summary

- Executable languages encourage precision and automation
- The language should support an appropriate model
  - Makes specication easy
- Strongly parallels programming languages
  - Structured vs. assembly languages
  - Object-oriented model and C++

Translation

- Model often unsupported by a standard language

(1) Use a **standard** language anyway
- Many tools available
- But, captures model unnaturally

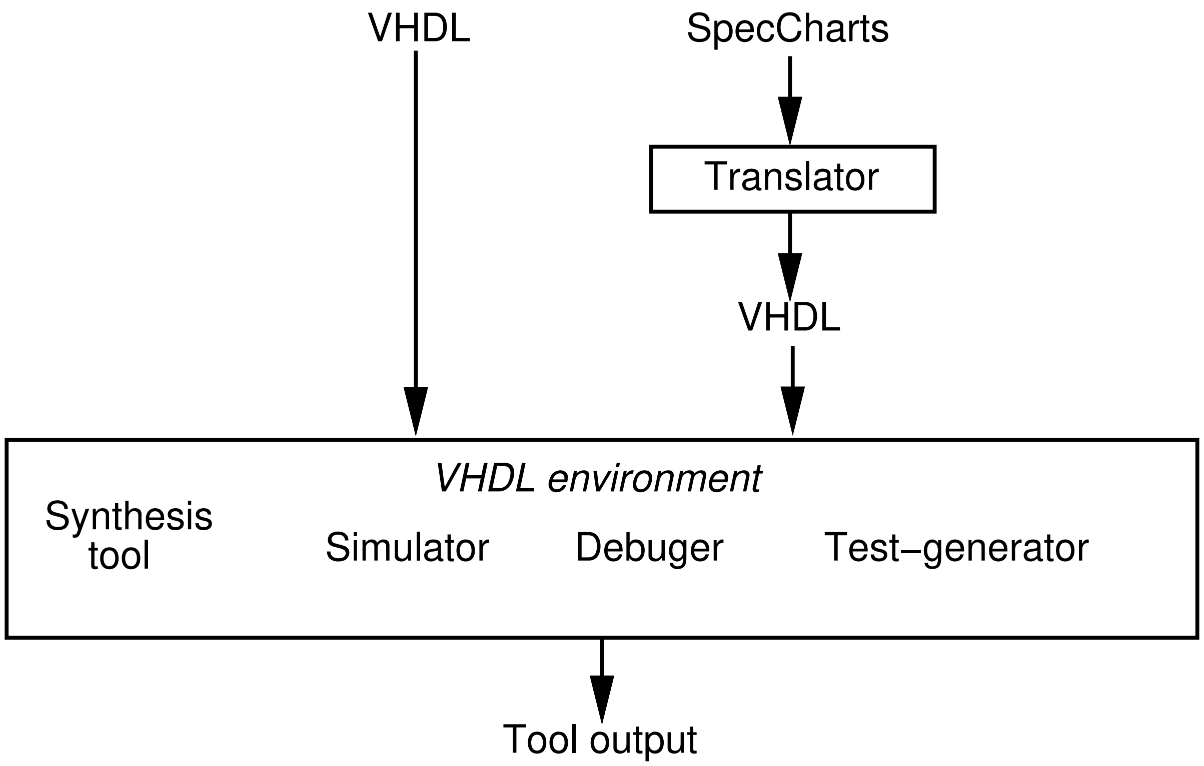
(2) Use an **application-specic** language
- Captures model naturally
- But, not many tools available

(3) Use a **front-end** language
- Captures model naturally
- Many tools available after **translating** to a standard

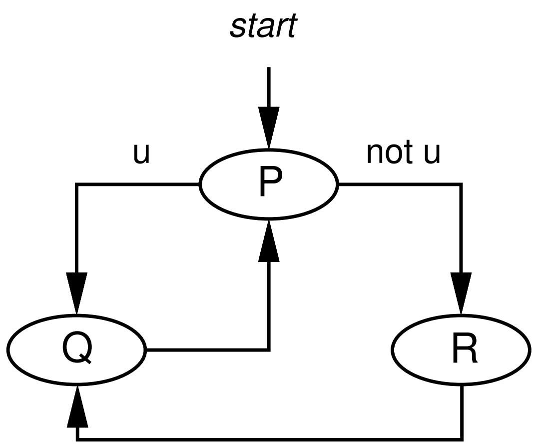
Outline

- Front-end language in VHDL environment
- State machine translation
- Fork-join translation
- Exception translation

A front-end language in a VHDL environment



State machine translation



(a)

```
type  state_type is (P, Q, R);
variable state : state_type := P;

loop
    case (state) is
        when P =>
            <actions for P>
            if (u) then
                state := Q;
            else if (not u) then
                state := R;
            end if;
        when Q =>
            <actions for Q>
            state := P;
        when R =>
            <actions for R>
            state := Q;
    end case;
end loop;
```

(b)

Fork-join translation

```
Main: process
begin

   statement1;

   parallel
   {
      P1;
      P2;
   }

   statement2;
   ...
```

(a)

```
signal fork, P1_done, P2_done : boolean;
```

```
Main : process
begin

   statement1;


   fork <= true;
   wait until P1_done
         and P2_done;


   statement2;
   ...
```

```
P1_process : process
begin

   wait until fork;

   P1;

   P1_done <= true;
   wait until not fork;
   P1_done <= false;

end;
```

(b)

Exception translation

```
event e :  T --> S;

  T :
     statement1;
     statement2;
     statement3;

  S :
     statement4;
     statement5;
```

(a)

```
          -- T
          statement1;
          if (e)
             goto S_start;
          statement2;
          if (e)
             goto S_start;
          statement3;

S_start:  -- S
          statement4;
          statement5;
```

(b)

```
-- T
T_loop : loop
   statement;
   if (e)
      exit T_loop;
   statement2;
   if (e)
      exit T_loop;
   statement3;
   exit T_loop;
end loop;

-- S
statement4;
statement5;
```

(c)

Summary

- The perfect standard language may never exist
- No standard language supports all models
- Using a front-end language solves the problem
  - Natural capture
  - Large base of tools and expertise
- Translators are simple
  - Maps characteristics to existing constructs
  - Generates well-structured and consistent output

System partitioning

- System functionality is implemented on system components
  - ASICs, processors, memories, buses
- Two design tasks:
  - **Allocate** system components or ASIC constraints
  - **Partition** functionality among components
- Constraints
  - Cost, performance, size, power
- Partitioning is a central system design task

Outline

- Structural vs. functional partitioning
- Natural vs. executable language specications
- Basic partitioning issues and algorithms
- Functional partitioning techniques for hardware
- Hardware/software partitioning
- Functional partitioning techniques for software
- Exploring tradeoffs with functional partitioning

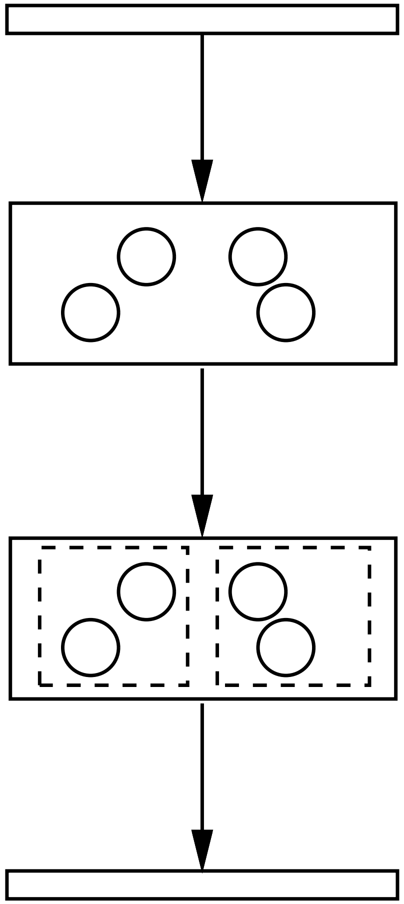
Structural vs. functional partitioning

- Structural: Implement structure, then partition
- Functional: Partition function, then implement
  - Enables better size/performance tradeoffs
  - Uses fewer objects, better for algorithms/humans
  - Permits hardware/software solutions
  - But, it's harder than graph partitioning

Natural vs. executable language specications

- Alternative methods for specifying functionality
- Natural languages common in practice
- Executable languages becoming popular
  - Automated estimation/partitioning explores solutions
  - Early verication reduces costly late changes
  - Precision eases integration

Basic partitioning issues



Specification abstraction−level

Granularity

Metrics and estimations

Partitioning algorithms

Objective and closeness functions

System−component allocation

Output

Basic partitioning issues (cont.)

- Specication-abstraction level: input denition
  - Just indicating the language is insufcient
  - Abstraction-level indicates amount of design already done
  - e.g. task DFG, tasks, CDFG, FSMD

- Granularity: specication size in each object
  - Fine granularity yields more possible designs
  - Coarse granularity better for computation, designer interaction
  - e.g. tasks, procedures, statement blocks, statements

- Component allocation: types and numbers
  - e.g. ASICs, processors, memories, buses

- Output: format and uses
  - e.g. new specication, hints to synthesis tool

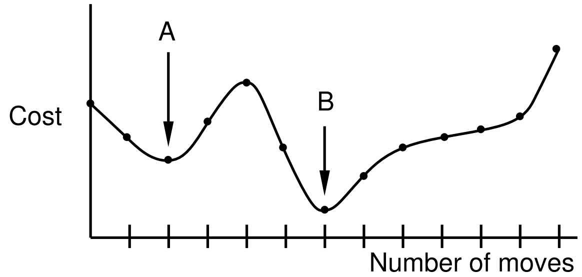
Basic partitioning issues (cont.)

- Metrics and estimations: "good" partition attributes
  - e.g. cost, speed, power, size, pins, testability, reliability
  - Estimates derived from quick, rough implementation
  - Speed and accuracy are competing goals of estimation

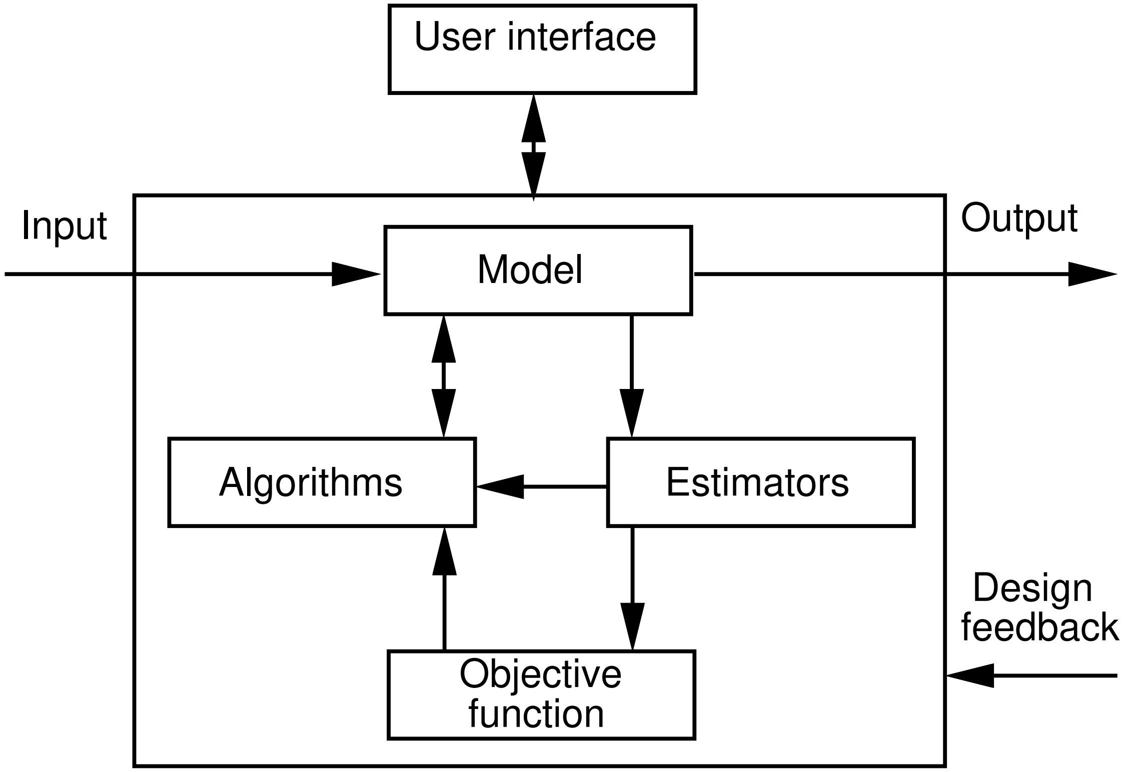
- Objective and closeness functions
  - Combines multiple metric values
  - Closeness used for grouping before complete partition
  - Weighted sum common
  - e.g. $k_1 F(area, c) + k_2 F(delay, c) + k_3 F(power, c)$

Basic partitioning issues (cont.)

- Algorithms: control strategies seeking best partition
  - Constructive creates partition
  - Iterative improves partition
  - Key is to escape local minimum



Typical partitioning-system conguration



Basic partitioning algorithms

- Clustering and multi-stage clustering [Joh67, LT91]
- Group migration (a.k.a. min-cut or Kernighan/Lin) [KL70, FM82]
- Ratio cut [KC91]
- Simulated annealing [KGV83]
- Genetic evolution
- Integer linear programming

Hierarchical clustering

- Constructive algorithm using closeness metrics

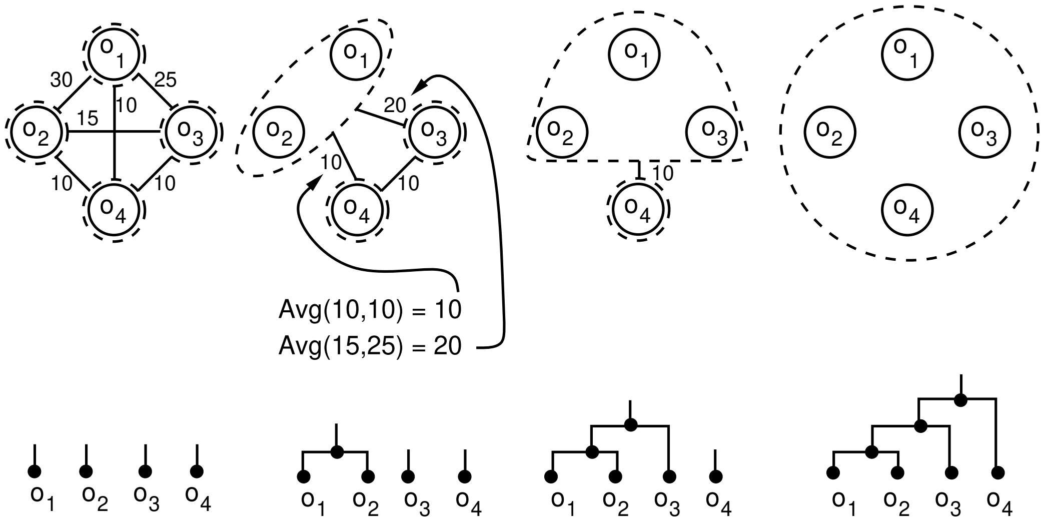
- Overview
  - Groups closest objects
  - Recomputes closenesses
  - Repeats until termination condition met

- Cluster tree maintains history of merges
  - Cutline across the tree denes a partition

Hierarchical clustering algorithm

```
/* Initialize each object as a group */
for each o_i loop
   p_i = o_i
   P = P ∪ p_i
end loop

/* Compute closenesses between objects */
for each p_i loop
   for each p_j loop
      c_{i,j} = ComputeCloseness(p_i, p_j)
   end loop
end loop

/* Merge closest objects and recompute closenesses
*/
while not Terminate(P) loop
   p_i, p_j = FindClosestObjects(P, C)
   P = P − p_i − p_j ∪ p_ij
   for each p_k loop
      c_{ij,k} = ComputeCloseness(p_ij, p_k)
   end loop
end loop

return P
```

Hierarchical clustering example



(a) (b) (c) (d)

Simulated annealing

- Iterative algorithm modeled after physical annealing process
- Overview
  - Starts with initial partition and temperature
  - Slowly decreases temperature
  - For each temperature, generates random moves
  - Accepts any move that improves cost
  - Accepts some bad moves, less likely at low temperatures
- Results and complexity depend on temperature decrease rate

Simulated annealing algorithm

```
temp = initial temperature
cost = Objfct(P)
while not Frozen loop
   while not Equilibrium loop
      P_tentative = Move(P)
      cost_tentative = Objfct(P_tentative)
      cost = cost_tentative − cost
      if (Accept(cost, temp) > Random(0, 1)) then
         P = P_tentative
         cost = cost_tentative
      end if
   end loop
   temp = DecreaseTemp(temp)
end loop
```

where: $Accept(cost, temp) = min(1, e^{-\frac{cost}{temp}})$

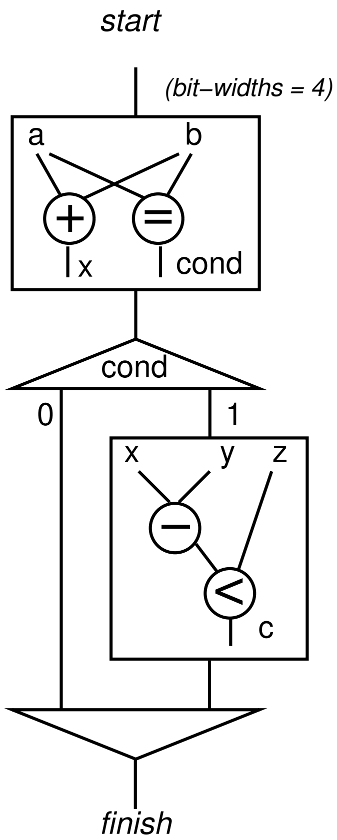
Functional partitioning for hardware: BUD

- Goal: incorporate area/time into synthesis [MK90]
- Clusters CDFG operations into datapath modules
- Closeness metrics:
  - Interconnecting wires
  - Concurrency
  - Shared hardware
- Each clustering corresponds to an allocation/scheduling
- Selects clustering with best area/time

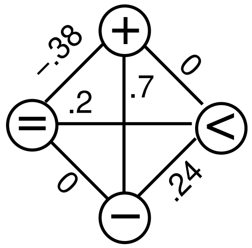
BUD example

```
x := a + b;
if (a = b)
   c := ((x – y) < z);
```

(a)

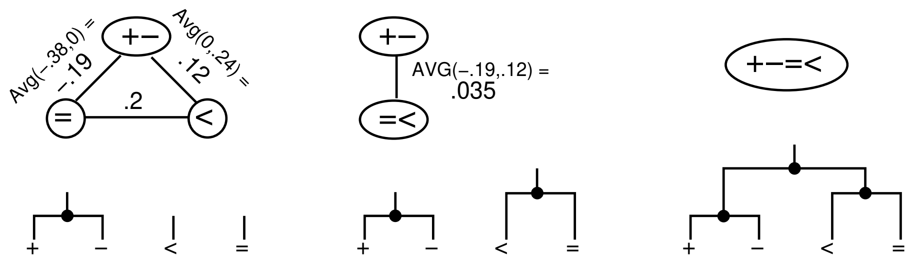


(b)



(c)

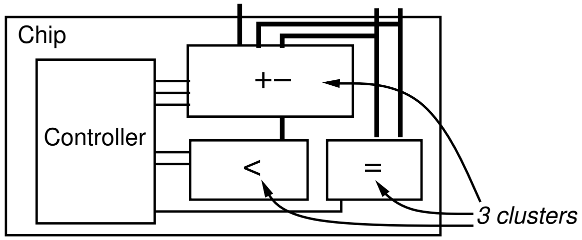
BUD example (cont.)



(a)

Clusters	Chip area A	Expected cycle time T	Objfct = AxT
+−=<	17.5	36	630
+−, =<	15.8	26	411
+−, =, <	13.8	26	359 (best)
+, −, =, <	16.4	26	426

(b)

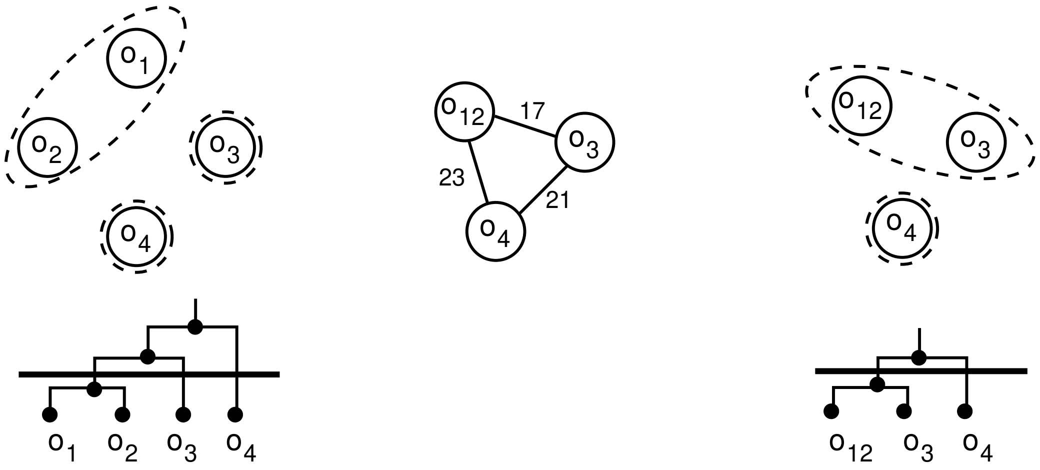


(c)

Functional partitioning for hardware: Aparty

- Extends BUD clustering to multiple stages [LT91]
  - Different closeness metrics for each stage
- Closeness metrics:
  - Control transfer reduction
  - Data transfer reduction
  - Hardware sharing

Aparty example



(a) **(b)** **(c)**

Hardware/software partitioning

- Combined hardware/software systems are common
- Software is cheap, modiable, and quick to design
- Hardware is fast
- Special algorithms are needed to favor software
- Proposed algorithms
  - Greedy [GD92]
  - Hill climbing [EHB94]
  - Binary-constraint search with hill climbing [VGG93]

Functional partitioning for systems: Vulcan, Cosyma

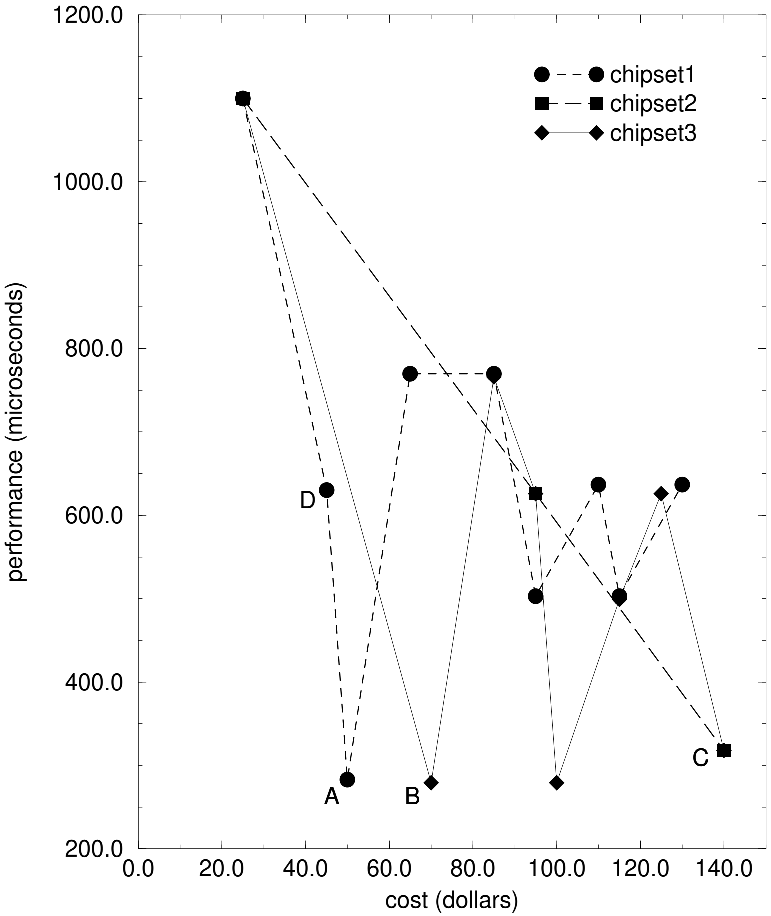
- Vulcan [GD90]I
  - Partitions CDFG operations among hardware only
  - Group migration and simulated annealing algorithms
- Vulcan II [GD93]
  - Partitions operations among hardware/software
  - Architecture: processor, hardware, memory, bus
  - All communication through memory
  - Uses greedy algorithm, extracts behaviors from hardware
- Cosyma [EHB94]
  - Partitions statement blocks among hardware/software
  - Architecture: processor, hardware, memory, bus
  - Simulated annealing, extracts behaviors from software

Functional partitioning for systems: SpecSyn

- Solves three partitioning problems
  - Behaviors to processors/ASICs
  - Variables to memories
  - Communication channels to buses
- Uses fast incremental-update estimators
- Covers both hardware and hardware/software partitioning [GVN94, VG92]

Exploring tradeoffs with functional partitioning

- Each line represents a different vendor's chip set
- Each point represents an allocation and partition
- Many designs quickly examined



Summary

- Partitioning heavily inuences design quality
- Functional partitioning is necessary
- Executable specication enables:
  - Automation
  - Exploration
  - Documentation
- Variety of algorithms exist
- Variety of techniques exist for different applications

Future directions

- Metrics from real design to guide partitioning
- Comparison of functional partitioning algorithms
- Impact of metric selections and orderings
- Impact of of granularity on partition quality
- Exploitation of regularity in partitioning

Estimation

- Estimates allow
  - Evaluation of design quality
  - Design space exploration
- Design model
  - Represents degree of design detail computed
  - Simple vs. complex models
- Issues for estimation
  - Accuracy
  - Speed
  - Fidelity

UC Irvine

Outline

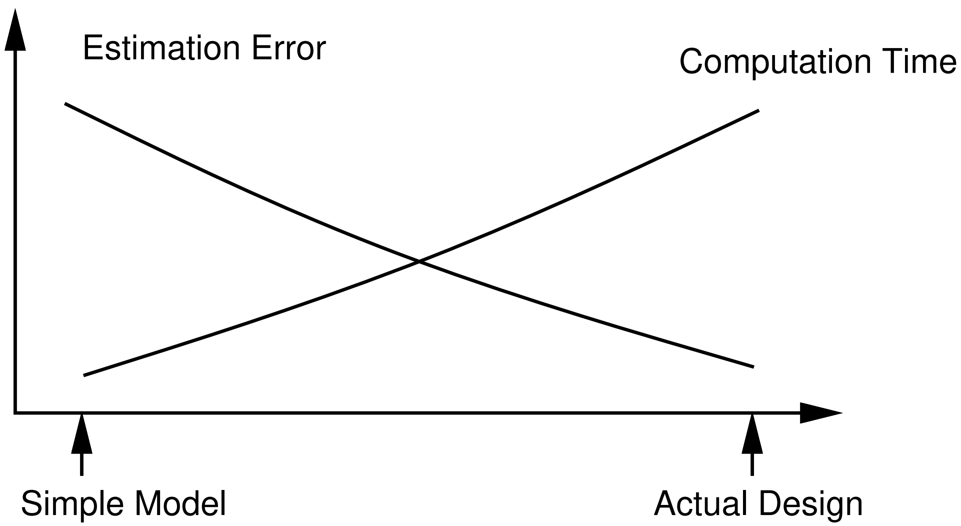
- Accuracy versus speed
- Fidelity
- Quality metrics
  - Performance metrics
  - Hardware and software cost metrics

Accuracy vs. Speed

- Accuracy: difference between estimated and actual value

$$\mathcal{A} = 1 - \frac{\mid E(D) - M(D) \mid}{M(D)}$$

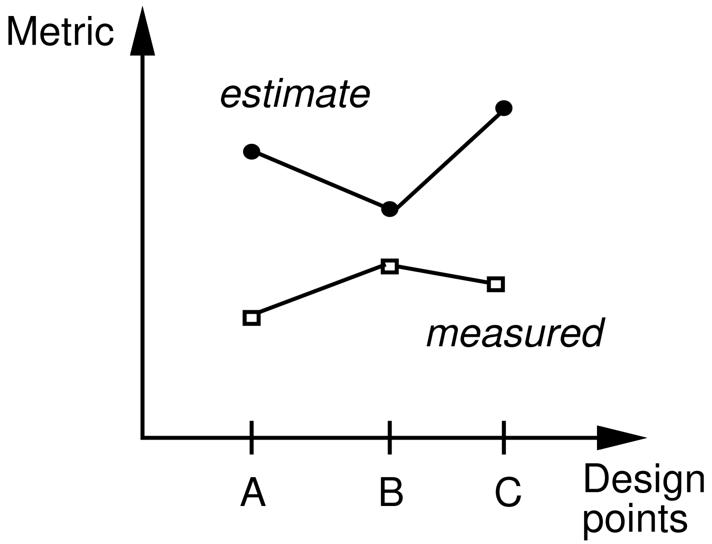
- Speed: computation time for obtaining estimate



UC Irvine

Fidelity

- Estimates must predict quality metrics for different design alternatives
- Fidelity: % of correct predictions for pairs of design implementations
- Higher delity $\Longrightarrow$ correct decisions based on estimates



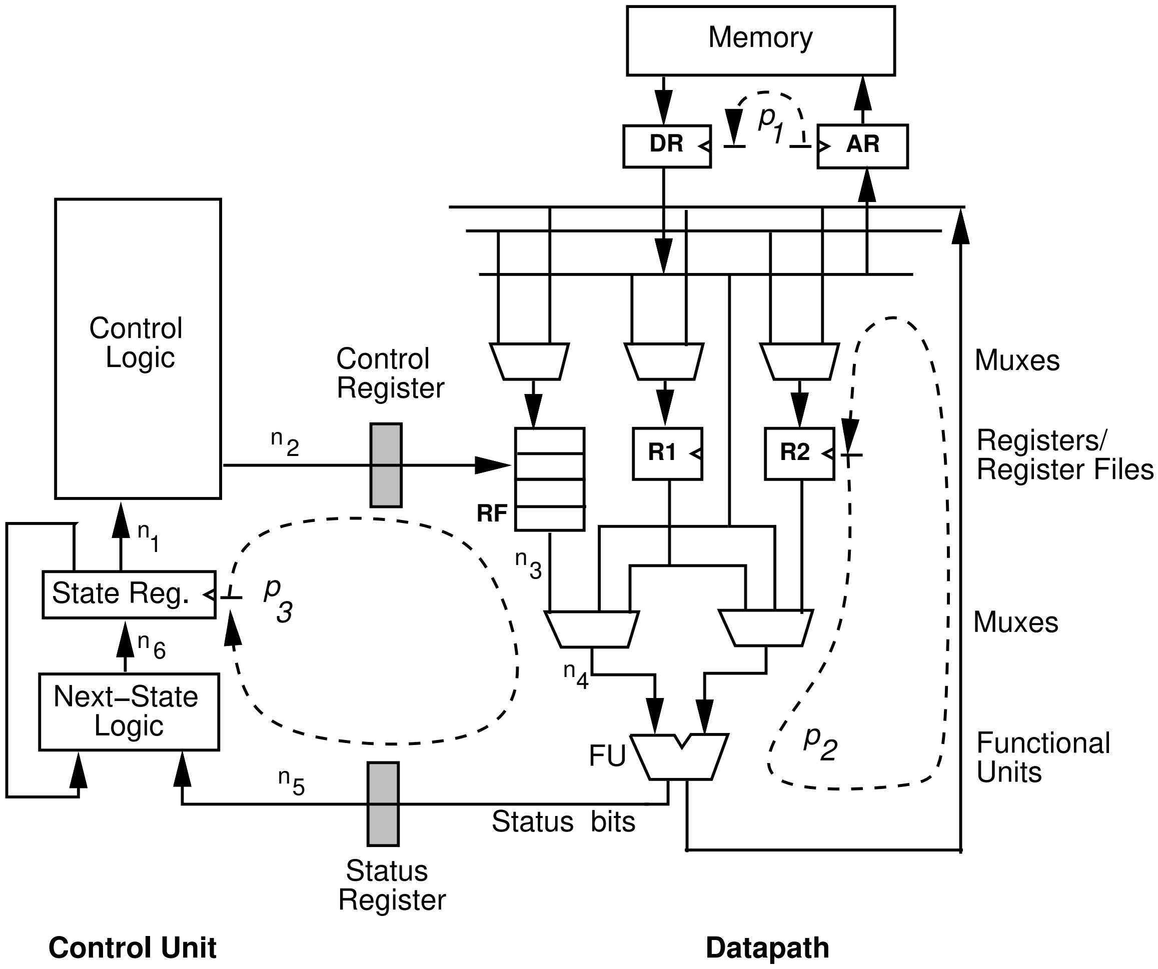
(A, B) =	E(A) > E(B), M(A) < M(B)	✗
(B, C) =	E(B) < E(C), M(B) > M(C)	✗
(A, C) =	E(A) < E(C), M(A) < M(C)	✓

Fidelity = 33 %

Quality metrics

- Performance Metrics
  - Clock cycle, control steps, execution time, communication rates
- Cost Metrics
  - **Hardware**: manufacturing cost (area), packaging cost(pin)
  - **Software**: program size, data memory size
- Other metrics
  - Power, testability, design time, time to market

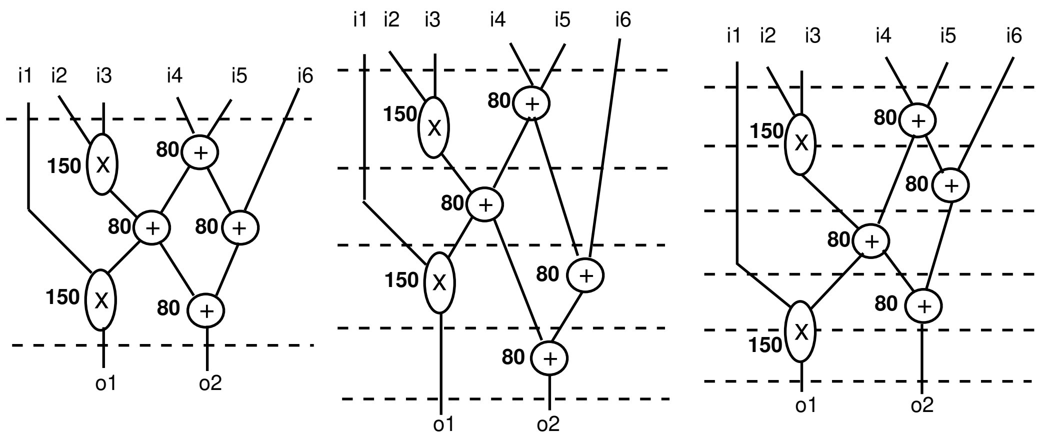
Hardware design model



Clock cycle estimation

- Clock cycle determines:
  - Resources, execution time

- Determining clock cycle
  - Designer specied [PK89, MK90]
  - Maximum delay of any functional unit [PPM86, JMP88]
  - Clock utilization [NG92]



Clock Cycle	: 380 ns	Clock Cycle	: 150 ns	Clock Cycle	: 80 ns
Exec. Time	: 380 ns	Exec. Time	: 600 ns	Exec. Time	: 400 ns
Resources	: 2 x, 4 +	Resources	: 1 x, 1 +	Resources	: 1 x, 1 +

Clock slack and utilization

- **Slack** : portion of clock cycle for which FU is idle

$$slack(clk, t_i) \; = \; (\; \lceil delay(t_i) \div clk \rceil \times clk \;) - delay(t_i)$$

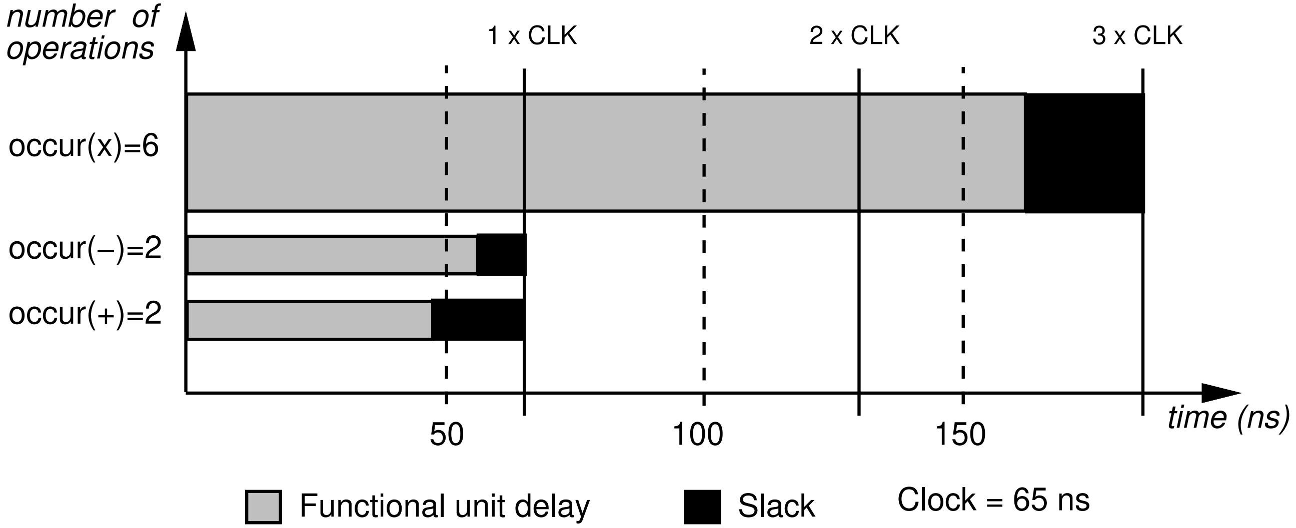
- **Average slack**: FU slack averaged over all operations

$$ave\_slack(clk) \; = \; \frac{\sum_{i}^{T} [\; occur(t_i) \times slack(clk, t_i) \;]}{\sum_{i}^{T} occur(t_i)}$$

- **Clock utilization** : % of clock cycle utilized for computations

$$utilization(clk) \; = \; 1 - \frac{ave\_slack(clk)}{clk}$$

Clock utilization



$$\text{ave\_slack}(65\text{ ns}) = \frac{\overset{6\text{x}32}{\boxed{\text{x}}} + \overset{2\text{x}9}{\boxed{-}} + \overset{2 \text{ x } 17}{\boxed{+}}}{6 + 2 + 2} = 24.4 \text{ ns}$$

$$\text{utilization}(65\text{ ns}) = 1 - (24.4 / 65.0) = 62 \%$$

UC Irvine

Slack minimization algorithm

Clock Slack Minimization [NG92]
Compute range: $clkmax, clkmin$
Compute occurrences: $occur(t_i)$

$max\_utilization = 0$
/* Examine each clock cycle in range */ **for** $clkmin \leq clk \leq clkmax$ **loop**

for all operation types $t_i \in T$ **loop**
Compute slack $slack(clk, t_i)$
end loop

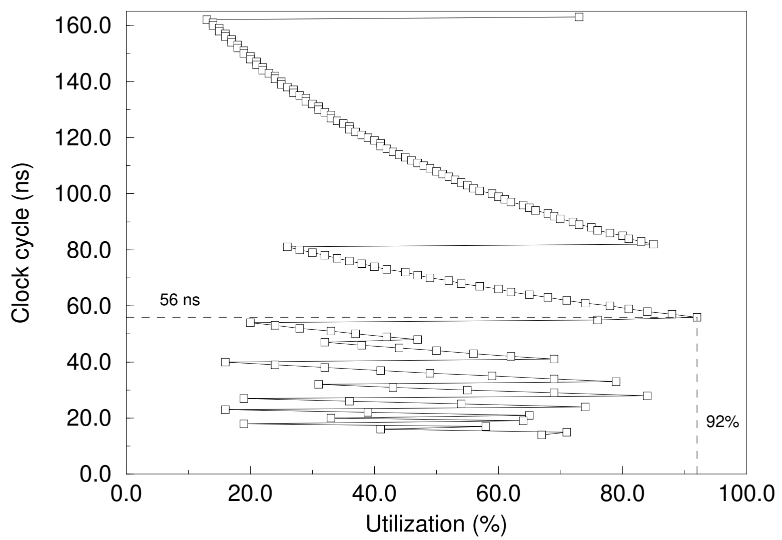
Compute average slack: $ave\_slack(clk)$
Compute utilization: $utilization(clk)$
/* If highest utilization */ **if** $utilization(clk) > max\_utilization$ **then**
$max\_utilization = utilization(clk)$
$max\_utilization\_clk = clk$
end if
end loop
$clk(SM) = max\_utilization\_clk$

Execution time vs. clock utilization

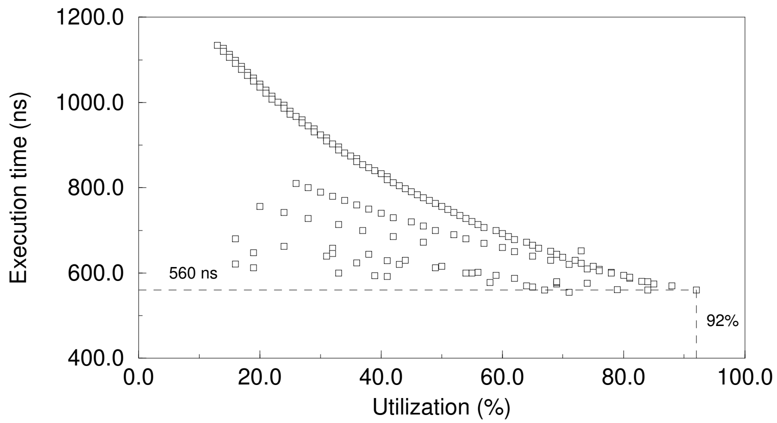
Second order differential equation example

- Clock with highest utilization results in better execution times

Clock cycle vs. Utilization



Execution time vs. utilization



Control steps estimation

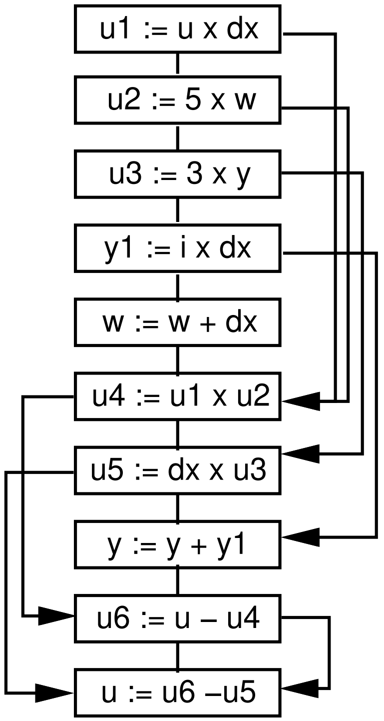
- Operations in the specication assigned to control step
- Number of **control steps** determines:
  - Execution time of design
  - Complexity of control unit
- Scheduling
  - Granularity is operations in a dataow graph
  - Computationally expensive

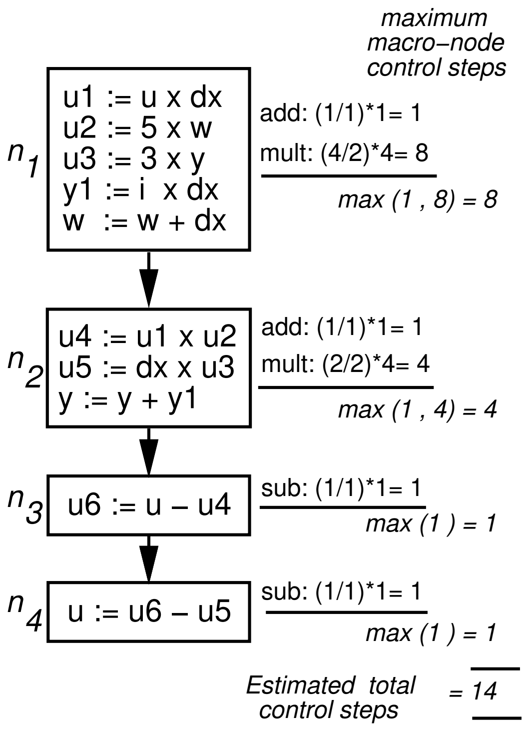
Operator-use method

- Granularity is statements in specication
- Faster than scheduling, average error 13%

t_i	$num(t_i)$	$clocks(t_i)$
add	1	1
mult	2	4
sub	1	1

u1 := u x dx ;
u2 := 5 x w ;
u3 := 3 x y ;
y1 := i x dx ;
w := w + dx ;
u4 := u1 x u2 ;
u5 := dx x u3 ;
y := y + y1 ;
u6 := u – u4 ;
u := u6 – u5 ;

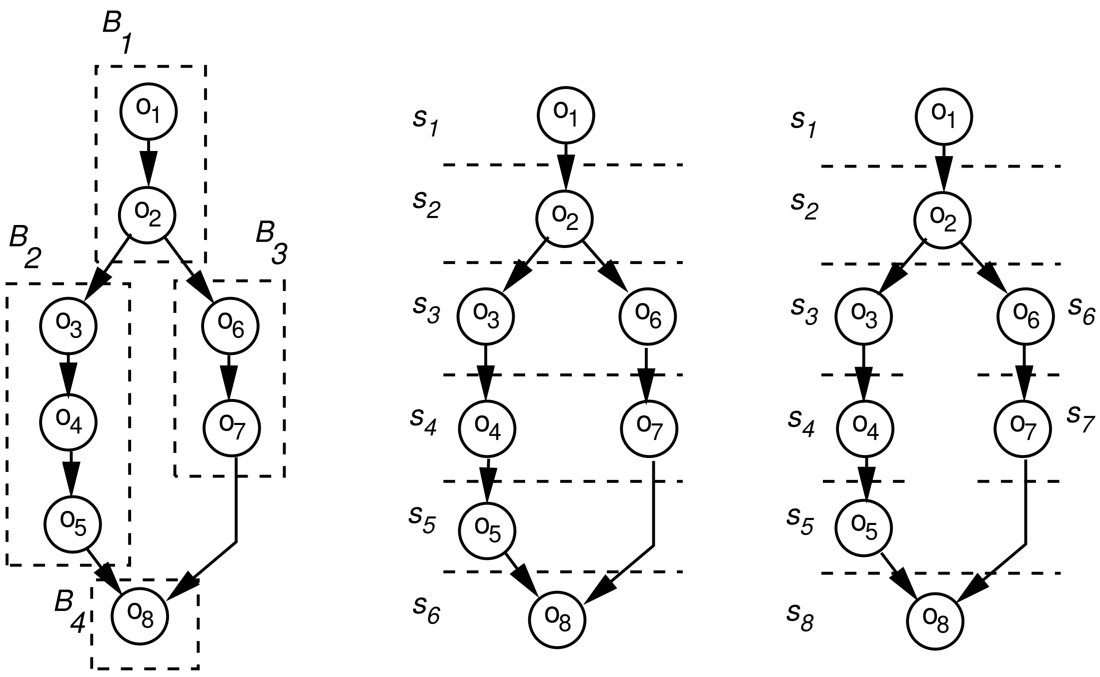




UC Irvine

Branching in behaviors

- Control steps maybe shared across exclusive branches
  - sharing schedule: fewer states, status register
  - non-sharing schedule: more states, no status registers



(a) (b) (c)

Execution time estimation

- Average start to nish time of behavior

- **Straight-line code behaviors**

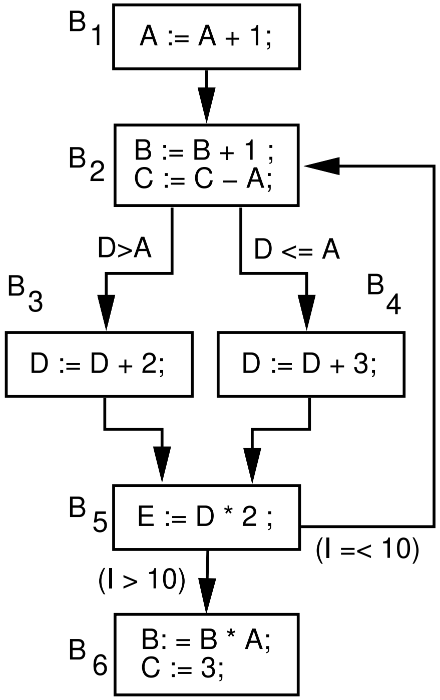
$$exectime(B) \;=\; csteps(B) \;\times\; clk$$

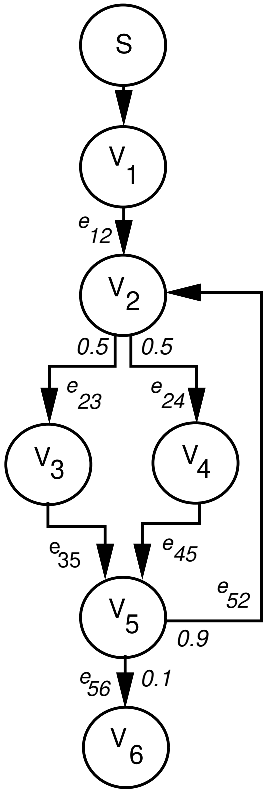
- **Behavior with branching**
  - Estimate execution time for each basic block
  - Create control ow graph from basic blocks
  - Determine branching probabilities
  - Formulate equations for node frequencies
  - Solve set of equations

$$exectime(B) = \sum_{b_i \in B} exectime(b_i) \times freq(b_i)$$

Probability-based ow analysis

```
A := A + 1;

for  I in 1 to 10  loop
     B := B + 1;
     C := C – A;

     if   (D  > A ) then
          D := D + 2;
     else
          D := D + 3;
     end if

     E := D * 2;
end loop;

B := B * A;
C := 3
```





Probability-based ow analysis

- Flow equations:

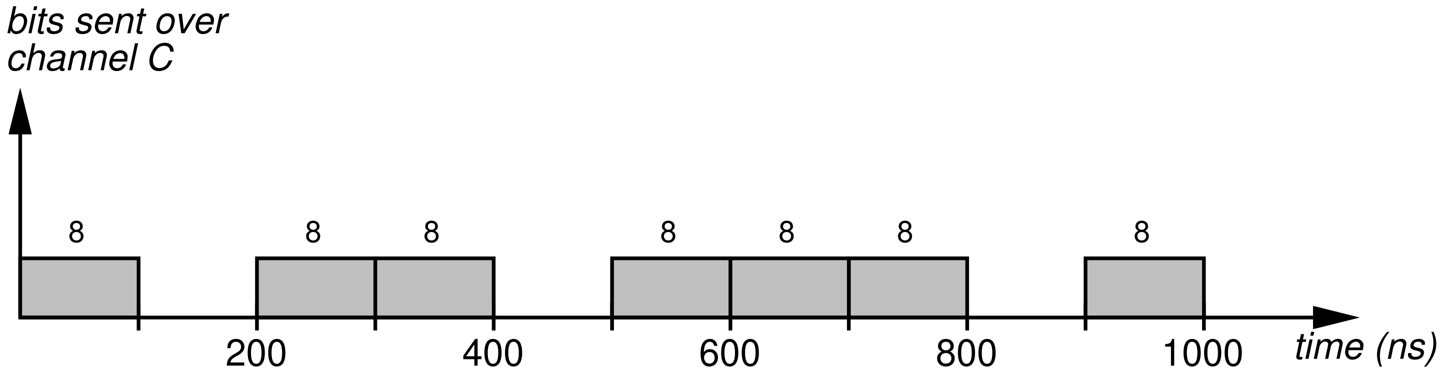
$$\begin{aligned}
freq(S) &= 1.0 \\
freq(v_1) &= 1.0 \times freq(S) \\
freq(v_2) &= 1.0 \times freq(v_1) + 0.9 \times freq(v_5) \\
freq(v_3) &= 0.5 \times freq(v_2) \\
freq(v_4) &= 0.5 \times freq(v_2) \\
freq(v_5) &= 1.0 \times freq(v_3) + 1.0 \times freq(v_4) \\
freq(v_6) &= 0.1 \times freq(v_5)
\end{aligned}$$

- Node execution frequencies:

$$\begin{aligned}
freq(v_1) &= 1.0 & freq(v_2) &= 10.0 \\
freq(v_3) &= 5.0 & freq(v_4) &= 5.0 \\
freq(v_5) &= 10.0 & freq(v_6) &= 1.0
\end{aligned}$$

- Can be used to estimate number of accesses to variables, channels or procedures

Communication rates



- **Average channel rate**

  rate of data transfer over lifetime of behavior

$$averate(C) = \frac{56\ bits}{1000\ ns} = 56\ Mb/s$$

- **Peak channel rate**

  rate of data transfer of single message

$$peakrate(C) = \frac{8\ bits}{100\ ns} = 80\ Mb/s$$

UC Irvine

Communication rate estimation

- Total behavior execution time consists of
  - Computation time, $comptime(P)$, obtained from ow-analysis
  - Communication time, $commtime(P, C) \;=\; access(P, C) \times delay(C)$
- Total bits transferred by the channel,

$$total\_bits(P, C) \;=\; access(P, C) \;\times\; bits(C)$$

- Channel average rate

$$averate(C) \;=\; \frac{total\_bits(B, C)}{comptime(B) + commtime(B, C)}$$

- Channel peak rate

$$peakrate(C) = \frac{bits(C)}{protocol\_delay(C)}$$

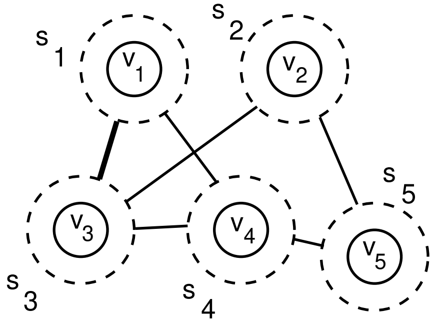
Area estimation

- Two tasks:
  - Determining number and type of components required
  - Estimating component size for a specic technology (FSMD, gate arrays etc.)
- Behavior implemented as a FSMD (nite state machine with datapath)
  - Datapath components: registers, functional units, multiplexers/buses
  - Control unit: state register, control logic, next-state logic
- We will discuss
  - Datapath component estimation
  - Control unit estimation
  - Layout area for a custom implementation

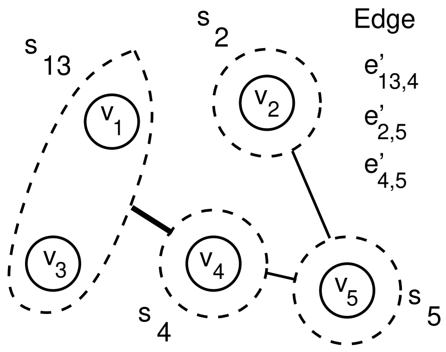
Clique-partitioning

- Commonly used for determining datapath components
- Let $G = (V, E)$ be a graph, V and E are set of vertices and edges
- Clique is a complete subgraph of G
- Clique-partitioning
  - divides the vertices into a minimal number of cliques
  - each vertex in exactly one clique
- One heuristic: maximum number of common neighbors [CS86]
  - Two nodes with maximum number of common neighbors are merged
  - Edges to two nodes replaced by edges to merged node
  - Process repeated till no more nodes can be merged

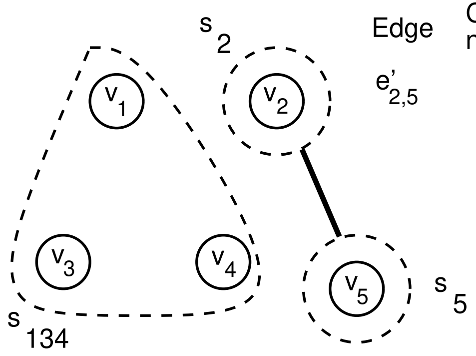
Clique-partitioning



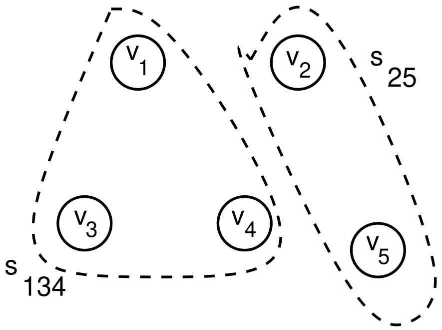
Edge	Common neighbors
$e'_{1,3}$	1
$e'_{1,4}$	1
$e'_{2,3}$	0
$e'_{2,5}$	0
$e'_{3,4}$	1
$e'_{4,5}$	0



Edge	Common neighbors
$e'_{13,4}$	0
$e'_{2,5}$	0
$e'_{4,5}$	0



Edge	Common neighbors
$e'_{2,5}$	0

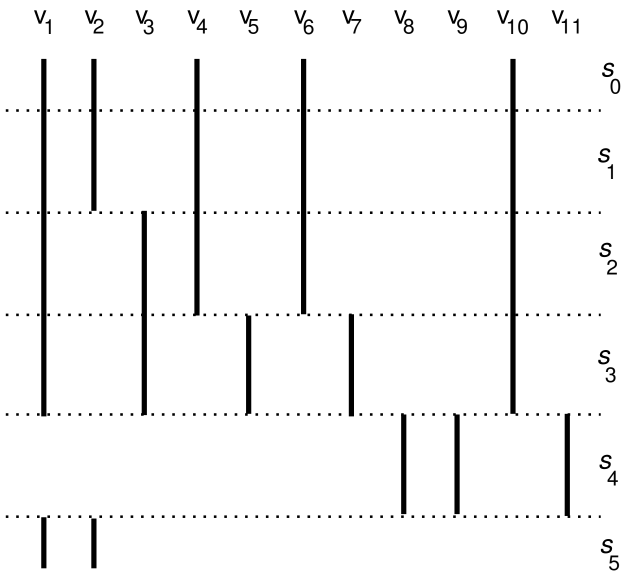


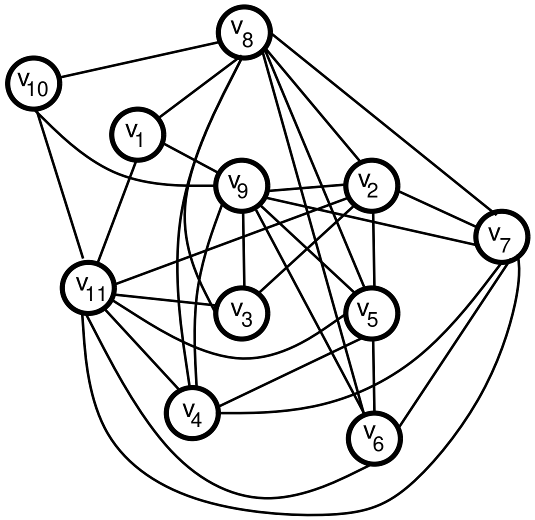
Cliques: $s_{134} = \{v_1, v_3, v_4\}$

$s_{25} = \{v_2, v_5\}$

Storage-unit estimation

- Variables not used concurrently maybe mapped same storage-unit
- To use clique-partitioning, construct a graph where
  - Each variable represented by a vertex
  - Variables with non-overlapping lifetimes have an edge between] their vertices





Cliques		Storage unit
$\{v_2, v_3\}$	=	R_1
$\{v_6, v_7, v_9\}$	=	R_2
$\{v_4, v_5, v_8\}$	=	R_3
$\{v_{10}, v_{11}\}$	=	R_4
$\{v_1\}$	=	R_5

Functional-unit and interconnect-unit estimation

- Clique-partitioning can be applied
- For determining the number of FU's required, construct a graph where
  - Each operation in behavior represented by a vertex
  - Edge connects two vertices if
    - Corresponding operations assigned different control steps
    - There exists an FU that can implement both operations
- For determining the number of interconnect units, construct a graph where
  - Each connection between two units is represented by a vertex
  - Edge connects two vertices if corresponding connections not used in same control step

Computing datapath area

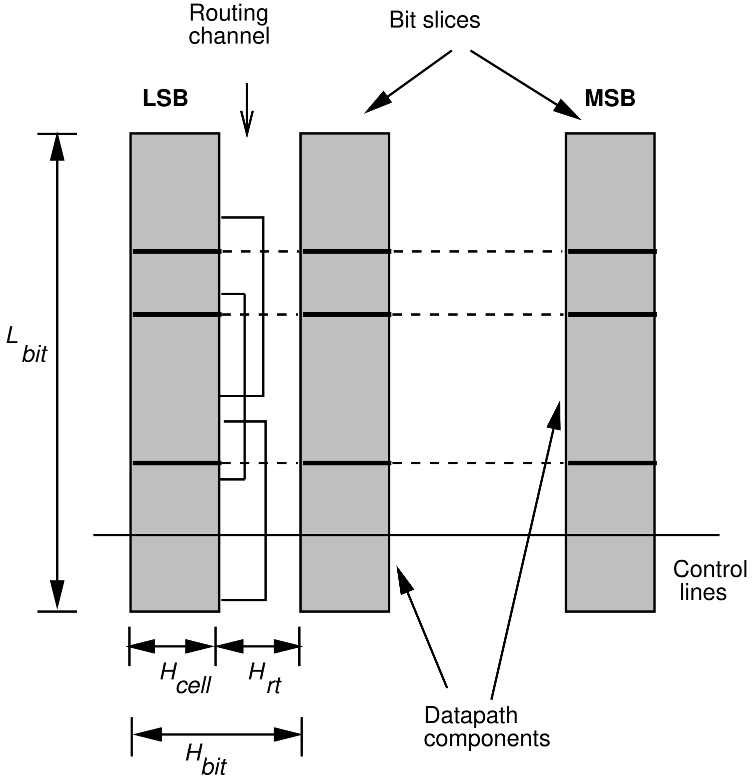
- Bit-sliced datapath

$$L_{bit} = \alpha \times tr(DP)$$

$$H_{rt} = \frac{nets}{nets\_per\_track} \times \beta$$

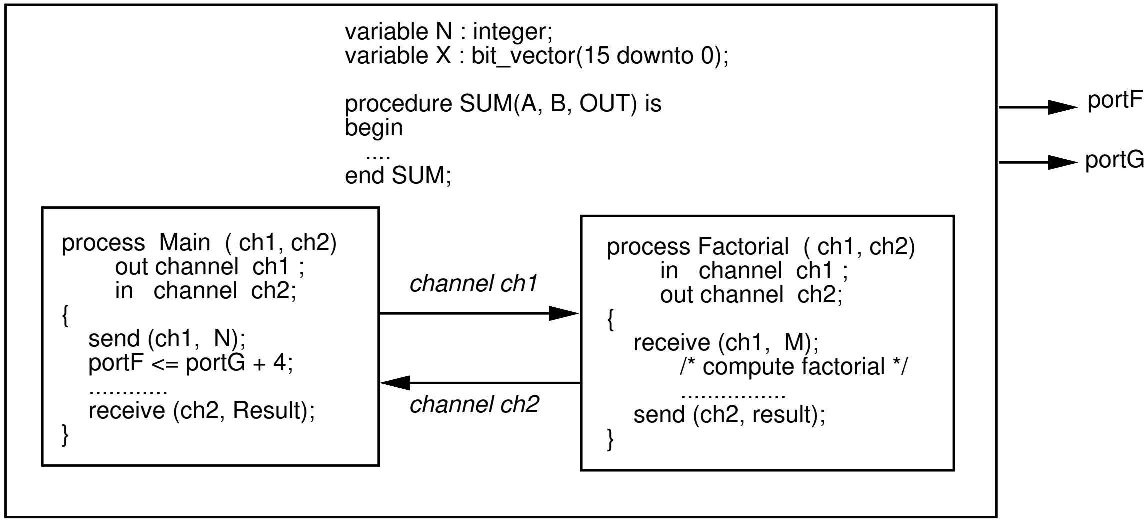
$$area(bit) = L_{bit} \times (H_{cell} + H_{rt})$$

$$area(DP) = bitwidth(DP) \times area(bit)$$

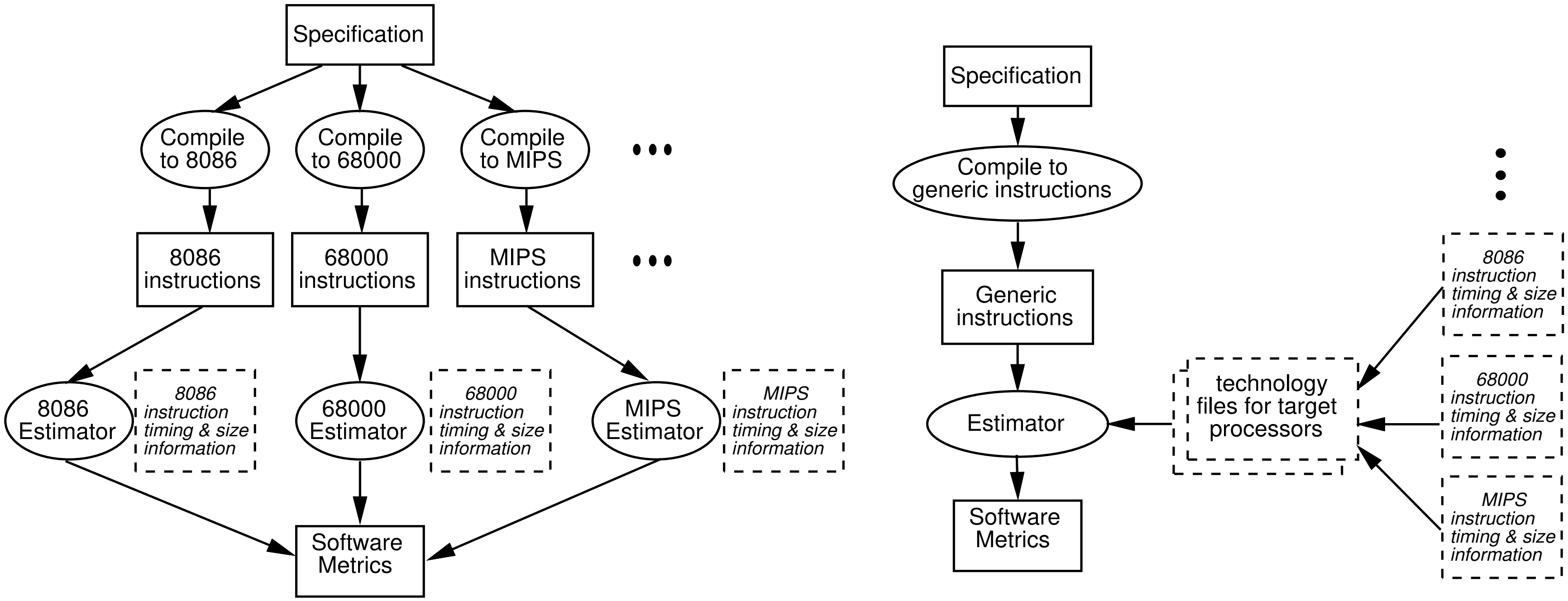


Pin estimation

- Number of wires at behavior's boundary depends on
  - Global data
  - Port accessed
  - Communication channels used
  - Procedure calls



Software estimation models



Processor specific model **Generic model**

Deriving processor technology les

Generic instruction

dmem3 = dmem1 + dmem2

8086 instructions

instruction	*clocks*	*bytes*
mov ax, word ptr[bp+offset1]	(10)	3
add ax, word ptr[bp+offset2]	(9 + EA1)	4
mov word ptr[bp+offset3], ax	(10)	3

68020 instructions

instruction	*clocks*	*bytes*
mov a6@(offset1), d0	(7)	2
add a6@(offset2), d0	(2 + EA2)	2
mov d0, a6@(offset3)	(5)	2

technology file for 8086

generic instruction	*execution time*	*size*
...		
dmem3 = dmem1 + dmem2	35 clocks	10 bytes
...		

technology file for 68020

generic instruction	*execution time*	*size*
...		
dmem3 = dmem1 + dmem2	22 clocks	6 bytes
...		

Software estimation

- **Program execution time**
  - Create basic blocks and compile into generic instructions
  - Estimate execution time of basic blocks
  - Perform probability-based ow analysis
  - Compute execution time of the entire behavior:

$$exectime(B) = \delta \times \left(\sum_{b_i \in B} exectime(b_i) \times freq(b_i) \right)$$

δ accounts for compiler optimizations

- **Program memory size**

$$progsize(B) \; = \; \sum_{g \in G} \; instr\_size(g)$$

- **Data memory size**

$$datasize(B) \; = \; \sum_{d \in D} datasize(d)$$

Summary and future directions

- We described methods for estimating:
  - Performance metrics: clock, control steps, execution time, communication rates
  - Cost metrics: design area, pins, program and data memory size

- Future directions:
  - Incorporating synthesis/compilation optimizations
  - New metrics for testability, power, integration cost, etc.
  - New architectural features for the estimation model

Renement

- Functional objects are grouped and mapped to system components
  - Functional objects: variables, behaviors, and channels
  - System components: memories, chips or processors, and buses
- **Renement** is update of specication to reect mapping
- Need for renement
  - Makes specication consistent
  - Enables simulation of specication
  - Generate input for synthesis, compilation and verication tools

Outline

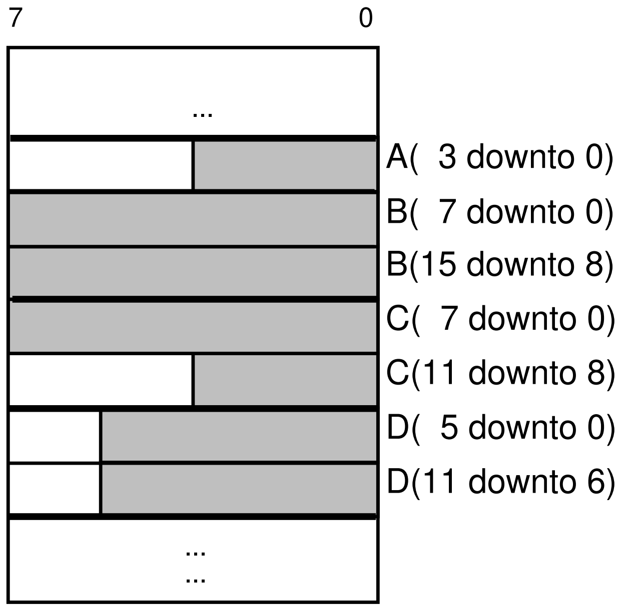
- Rening variable groups
- Channel renement
- Resolving access conicts
- Rening incompatible interfaces

Rening variable groups

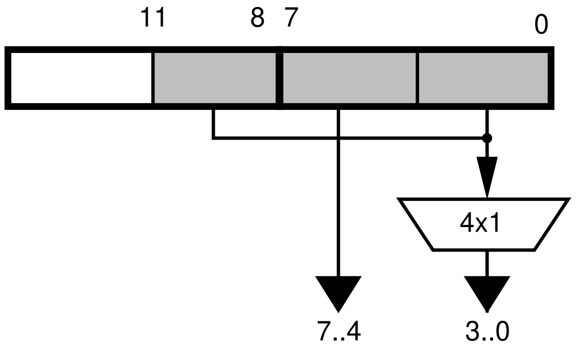
- Group of variables mapped to a memory
- Variable folding:
  - Implementing each variable in a memory with a xed word size
- Memory address translation
  - Assignment of addresses to each variable in group
  - Update references to variable by accesses to memory

Variable folding

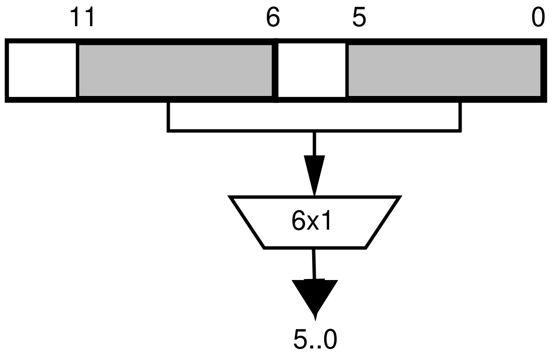
```
variable  A :  bit_vector(  3 downto 0) ;
variable  B :  bit_vector(15 downto 0) ;
variable  C :  bit_vector(11 downto 0) ;
variable  D :  bit_vector(11 downto 0) ;
```



8−bit Memory



to variable C in memory



to variable D in memory

Memory address translation

```
variable  J, K : integer := 0;
variable  V : IntArray (63 downto 0);
....
V(K) := 3;
X := V(36);
V(J) := X;
....
for  J  in 0 to 63 loop
    SUM := SUM + V(J);
end loop;
....
```

Original specification

V (63 downto 0)

↓

MEM(163 downto 100)

Assigning addresses to V

```
variable  J, K : integer := 0;
variable  MEM : IntArray (255 downto 0);
....
MEM(K +100) := 3;
X := MEM(136);
MEM(J+100) := X;
....
for  J in 0 to 63 loop
    SUM := SUM + MEM(J +100);
end loop;
....
```

Refined specification

```
variable  J : integer := 100;
variable  K : integer := 0;
variable  MEM : IntArray (255 downto 0);
....
MEM(K + 100) := 3;
X := MEM(136);
MEM(J) := X;
....
for  J  in 100 to 163 loop
    SUM := SUM + MEM(J);
end loop;
....
```

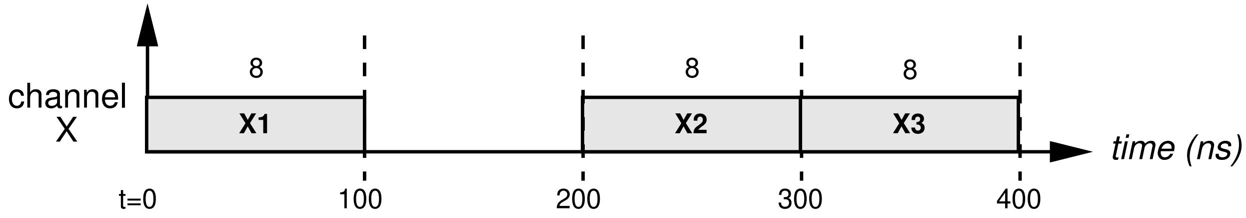
Refined specification
without offsets for index J

Rening channel groups

- Channels are virtual entities over which messages are transferred
- Bus is a physical medium that implements groups of channels
- Bus consists of:
  - wires representing data and control lines
  - protocol dening sequence of assignments to data and control lines
- Two renement tasks
  - *Bus generation*: determining buswidth i.e. number of data lines
  - *Protocol generation*: specifying mechanism of transfer over bus

Characterizing communication channels

- For a given behavior P that sends data over channel C,
  - **Message size**, $bits(C)$: number of bits in each message
  - **Accesses**, $accesses(P, C)$: number of times P transfers data over C
  - **Average rate**, $averate(C)$: rate of data transfer of C over lifetime of behavior
  - **Peak rate**, $peakrate(C)$: rate of transfer of single message



$$bits(C) = 8 \text{ bits}$$
$$averate(C) = \frac{24\ bits}{400\ ns} = 60\ Mbits/s$$
$$peakrate(C) = \frac{8\ bits}{100\ ns} = 80\ Mbits/s$$

Characterizing buses

- For a given bus B,
  - **Buswidth** , $buswidth(B)$: number of data lines in B
  - **Protocol delay**, $protdelay(B)$: delay for single message transfer over bus
  - **Average rate**, $averate(B)$: rate of data transfer over B over lifetime of system
  - **Peak rate**, $peakrate(B)$: maximum rate of transfer of data on bus

$$peakrate(C) = \frac{buswidth(B)}{protdelay(B)}$$

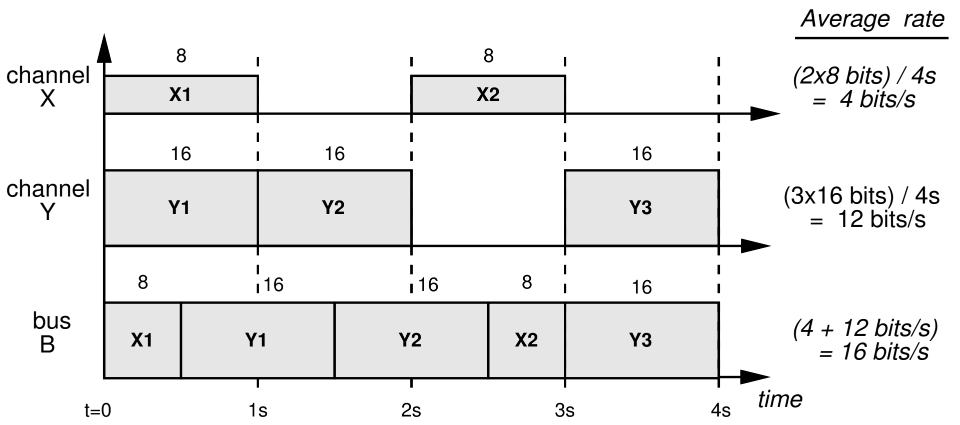
Determining bus rates

- Idle slots of a channel used for messages of other channels
- To ensure that channel average rates are unaffected by bus

$$averate(B) \geq \sum_{C \in B} averate(C)$$

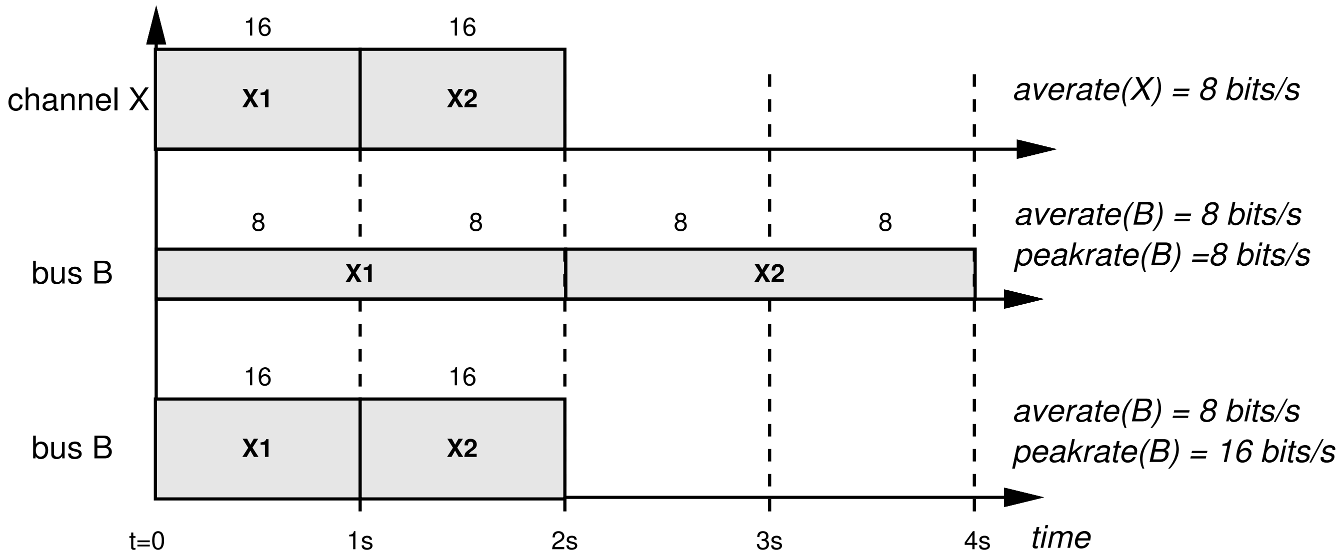
- Goal: to synthesize a bus that *constantly* transfers data i.e.

$$peakrate(B) = averate(C)$$



Constraints for bus generation

- **Buswidth**: affects number of pins on chip boundaries
- **Channel average rates**: affects execution time of behaviors
- **Channel peak rates**: affects time required for single message transfer



Bus generation algorithm [NG94]

```
/* Determine range of buswidths */
minwidth = 1,   maxwidth = Max(bits(C))

mincost = ∞,   mincostwidth = ∞
for currwidth in minwidth to maxwidth loop
    /* compute bus peak rate */
    peakrate(B) = currwidth ÷ protdelay(B)
    /* compute sum of channel average rates */
    averatesum = 0;
    for all channels C ∈ B loop
        averate(C) = (access(P, C) × bits(C)) / (comptime(P) + commtime(P))
        averatesum = averatesum + averate(C);
    end loop
    if (peakrate(B) > averatesum) then
        /* feasible solution, determine minimal cost */
        currcost = ComputeCost(currwidth)
        if (currcost < mincost) then
            mincost = currcost,   mincostwidth = currwidth
        end if
    end if
end loop
return(mincostwidth)
```

Bus generation algorithm

- **Compute buswidth range**: $minwidth = 1,\ maxwidth = \text{Max}(bits(C))$
- **For** $minwidth \leq currwidth \leq maxwidth$ **loop**
  - **Compute bus peak rate**:

    $$peakrate(B) = currwidth \div protdelay(B)$$

  - **Compute channel average rates**

    $$commtime(P) = access(P,C) \times [\ \lceil \tfrac{bits(C)}{currwidth} \rceil \times protdelay(B)\ ]$$

    $$averate(C) = \frac{access(P,C) \times bits(C)}{comptime(P) + commtime(P)}$$

  - **if** $peakrate(B) \geq \sum_{C \in B} averate(C)$ **then**

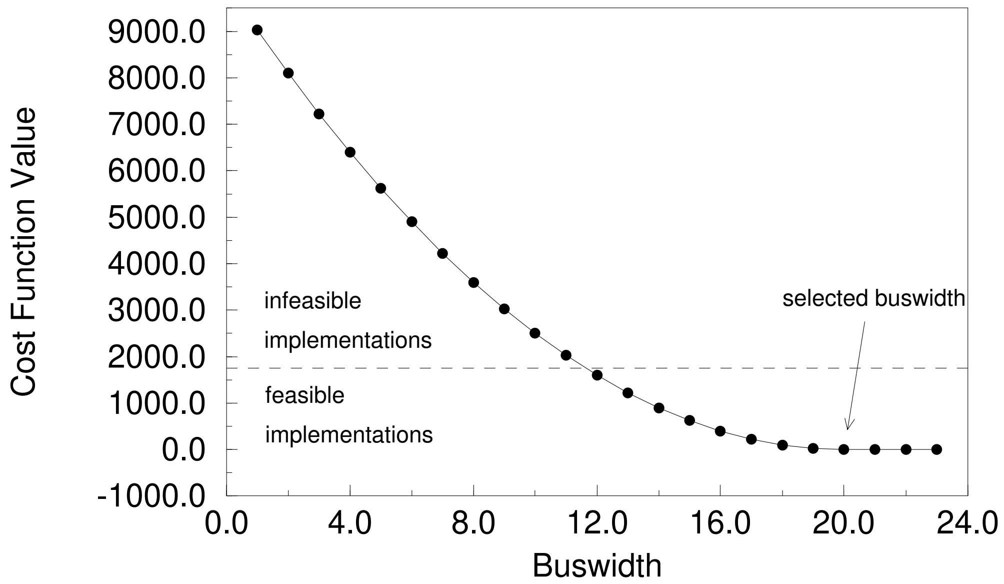
    if $bestcost > ComputeCost(currwidth)$ **then**

    $bestcost = ComputeCost(currwidth)$

    $bestwidth = currwidth$

Bus generation example

- 2 behavior accessing 16 bit data over two channels
- Constraints specied for channel peak rates

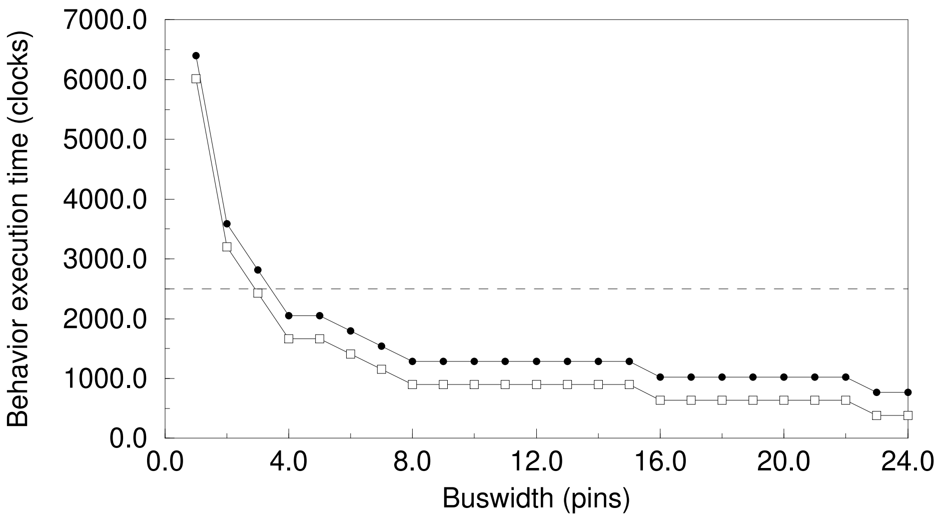


Performance vs. buswidth tradeoffs

- Allows a buswidth to be selected, given performance constraints

  e.g. behavior $P1$ has performance constraint of 2500 clocks.

  buswidths of 4 or greater must be selected

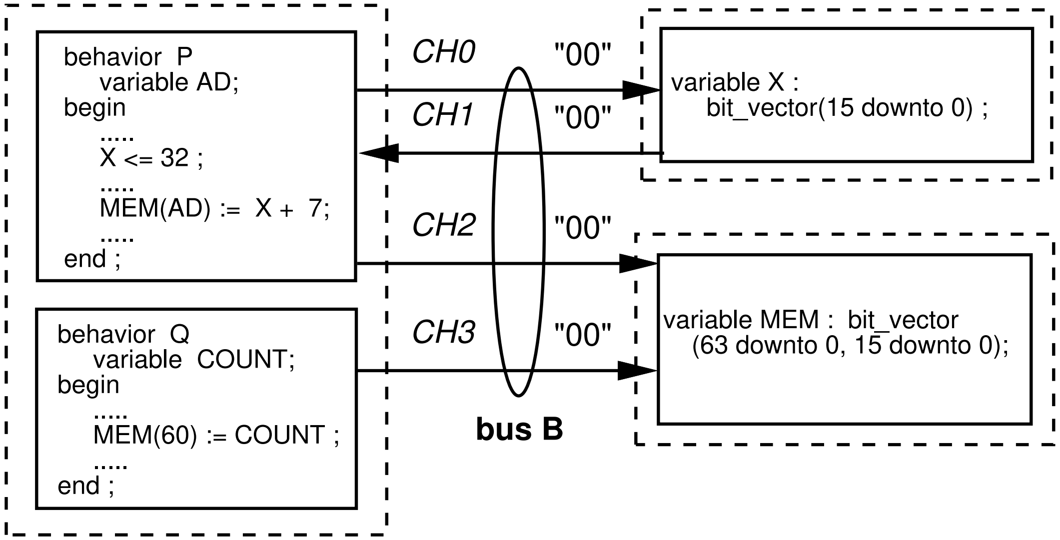


Protocol generation

- Bus consists of several sets of wires:
  - **Data lines**, used for transferring message bits
  - **Control lines**, used for synchronization between behaviors
  - **ID lines**, used for identifying the channel active on the bus
- All channels mapped to bus share these lines
- Number of data lines determined by bus generation algorithm
- Protocol generation consists of six steps

Protocol generation

1. **Protocol selection**: full handshake, half-handshake etc.
2. **ID assignment**: N channels require $log_2(N)$ ID lines



Protocol generation

3. Bus structure denition

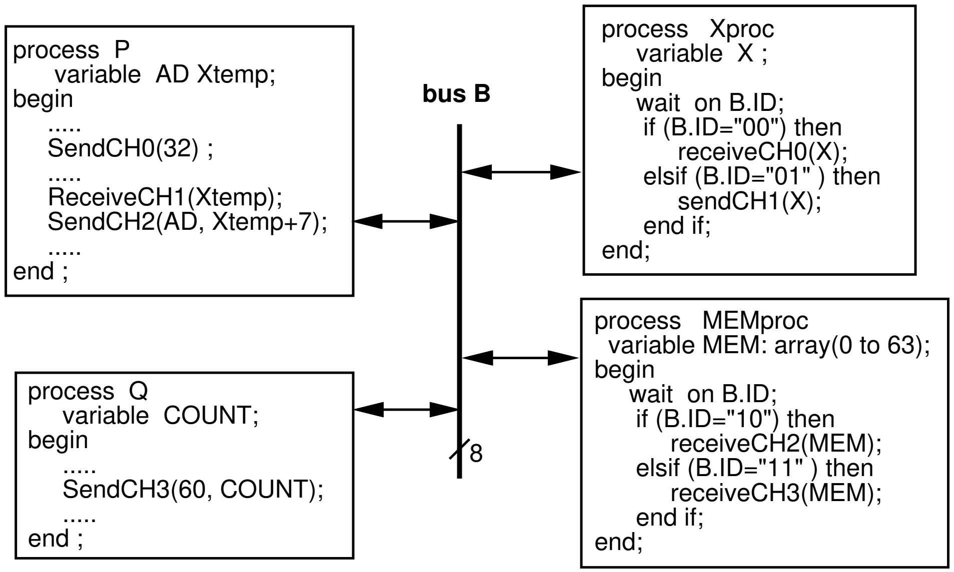
```
type  HandShakeBus is record
        START, DONE : bit ;
        ID : bit_vector(1 downto 0) ;
        DATA  : bit_vector(7 downto 0) ;
end record ;

signal  B : HandShakeBus ;
```

4. Bus protocol denition

```
procedure  ReceiveCH0( rxdata : out bit_vector)  is
begin
    for J in 1 to 2 loop
        wait  until  (B.START = '1') and (B.ID = "00") ;
        rxdata (8*J–1  downto 8*(J–1)) <= B.DATA ;
        B.DONE <= '1' ;
        wait until  (B.START = '0') ;
        B.DONE <= '0' ;
    end loop;
end ReceiveCH0;

procedure  SendCH0( txdata : in bit_vector)  is
begin
    bus B.ID <= "00" ;
    for J in 1 to 2 loop
        B.data <=  txdata(8*J–1  downto  8*(J–1)) ;
        B.START <= '1' ;
        wait until  (B.DONE = '1') ;
        B.START <= '0' ;
        wait until  (B.DONE = '0') ;
    end loop;
end SendCH0;
```

Protocol generation

5. **Update variable references**
6. **Generate behaviors for variables**



UC Irvine

Resolving access conicts

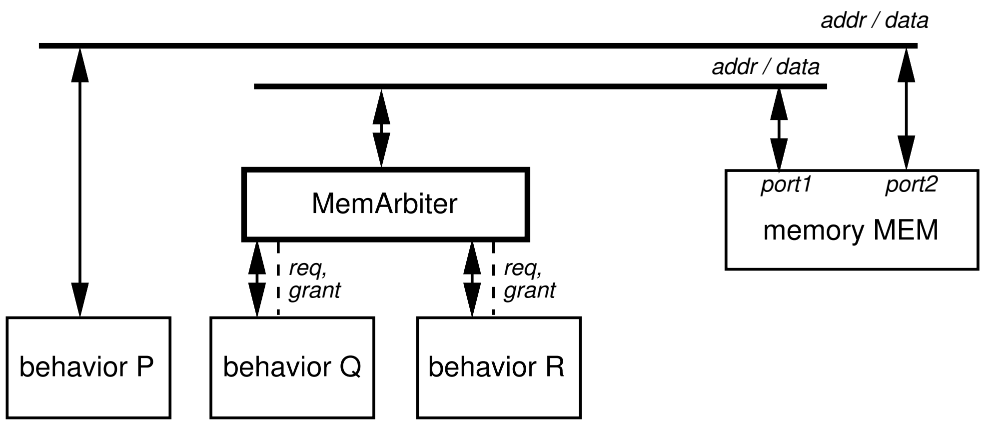
- System partitioning may result in concurrent accesses to a resource
  - Channels mapped to a bus may attempt data transfer simultaneously
  - Variables mapped to a memory may be accessed by behaviors simultaneously

- Arbiter needs to be generated to resolve such access conicts

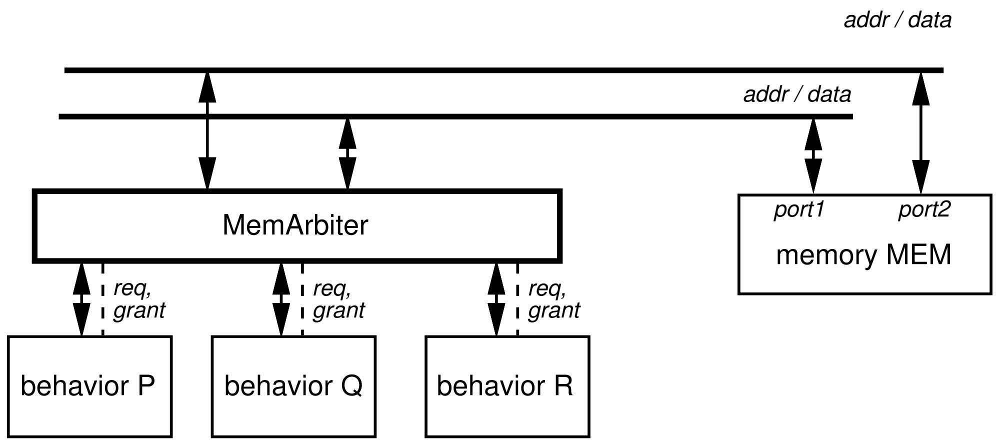
- Three tasks
  - Arbitration model selection
  - Arbitration scheme selection
  - Arbiter generation

Arbitration models

Static



Dynamic



Arbiter generation

- Example of bus arbitration
  - Two behaviors accessing a single resource, bus B
  - Behavior P assigned higher priority than Q
  - Fixed priority implemented with two handshake signals Req and $Grant$

```
process B_arbiter
begin
    wait until (Req_P='1')
            or (Req_Q = '1');
    if (Req_P = '1') then
        Grant_P = '1';
        wait unitl (Req_P = '0');
        Grant_P = '0";
    elsif (Req_Q = '1') then
        Grant_Q <= '1';
        wait until (Req_Q = '0');
        Grant_Q <= '0';
    end if;
end process;
```

Req_P
Grant_P

```
process P
    variable AD Xtemp;
begin
    .....
    Req_P <= '1';
    wait until (Grant_P = '1');
    SendCH0(32) ;
    Req_P <= '0';
    .....
end process ;
```

Req_Q
Grant_Q

```
process Q
    variable COUNT;
begin
    .....
    Req_Q <= '1';
    wait until (Grant_Q = '1');
    SendCH3(60, COUNT);
    Req_Q <= '0';
    .....
end process;
```

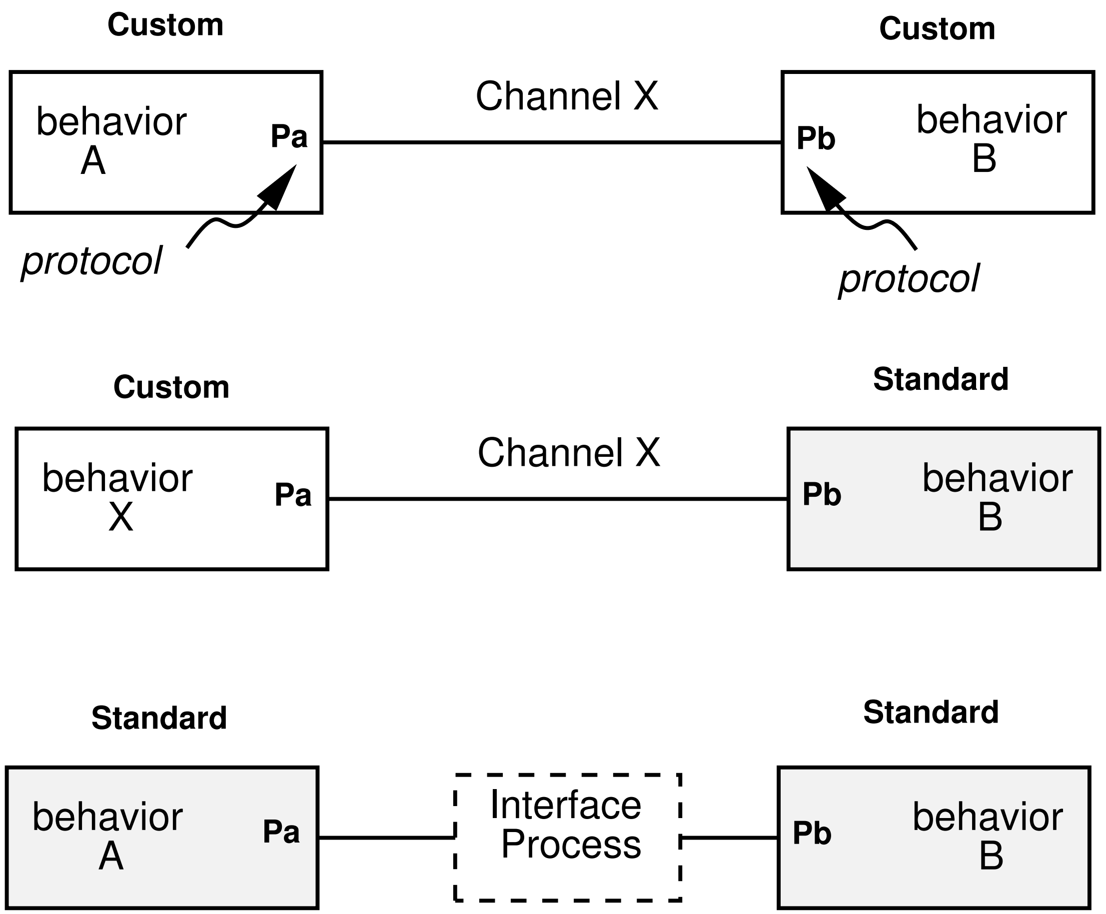
bus B

8

```
process Xproc
  variable X ;
begin
    wait on B.ID;
    if (B.ID="00") then
        receiveCH0(X);
    elsif (B.ID="01" ) then
        sendCH1(X);
    end if;
end process;
```

```
process MEMproc
  variable MEM: array(0 to 63);
begin
    wait on B.ID;
    if (B.ID="10") then
        receiveCH2(MEM);
    elsif (B.ID="11" ) then
        receiveCH3(MEM);
    end if;
end process;
```

Effect of binding on interfaces



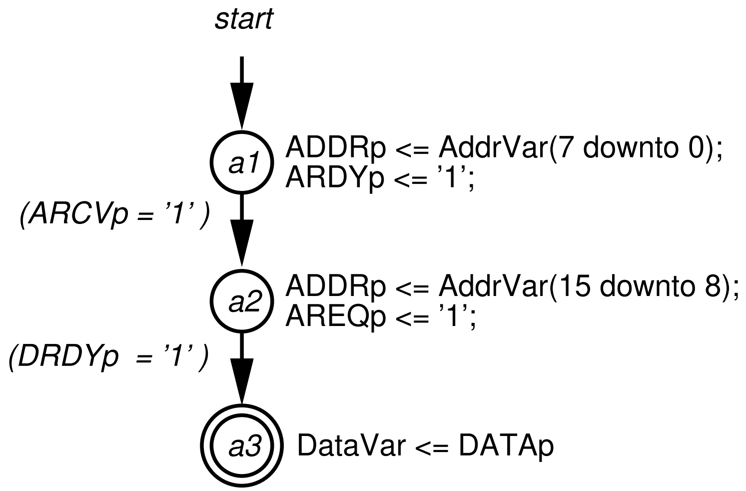
Protocol operations

- Protocols usually consist of ve *atomic* operations
  - waiting for an event on input control line
  - assigning value to output control line
  - reading value from input data port
  - assigning value to output data port
  - waiting for xed time interval

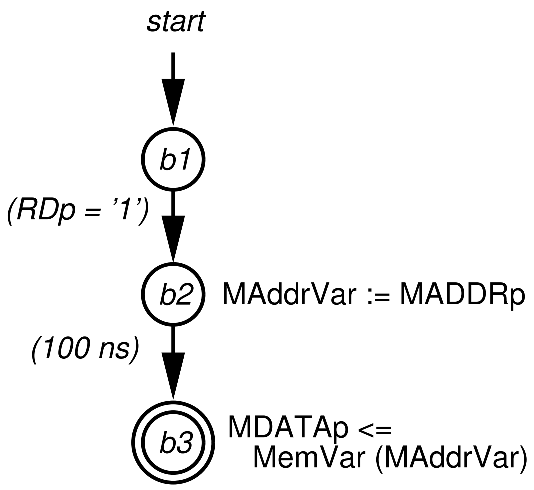
- Protocol operations may be specied in one of three ways
  - Finite state machines (FSMs)
  - Timing diagrams
  - Hardware description languages (HDLs)

Protocol specication : FSMs

- Protocol operations ordered by sequencing between states
- Constraints between events may be specied using timing arcs
- Conditional & repetitive event sequences require extra states, transitions



Protocol Pa

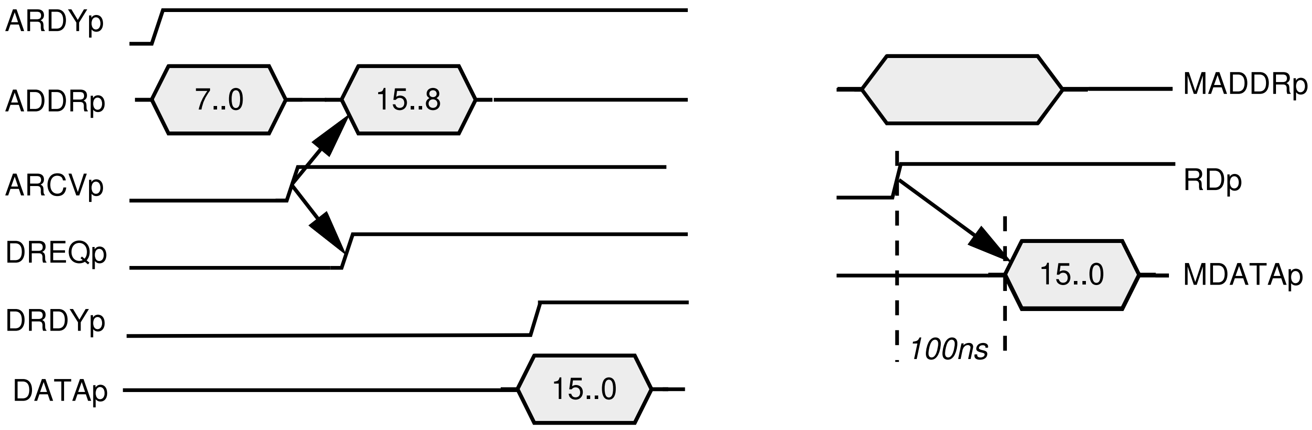


Protocol Pb

UC Irvine

Protocol specication : Timing diagrams

- Advantages:
  - Ease of comprehension, representation of timing constraints
- Disadvantages:
  - Lack of action language, not simulatable
  - Difcult to specify conditional and repetitive event sequences

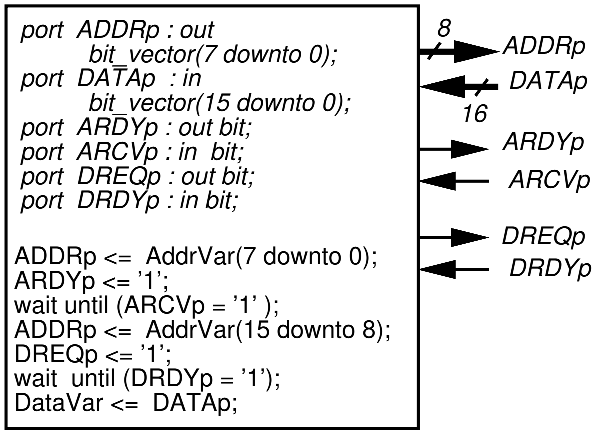


Protocol Pa **Protocol Pb**

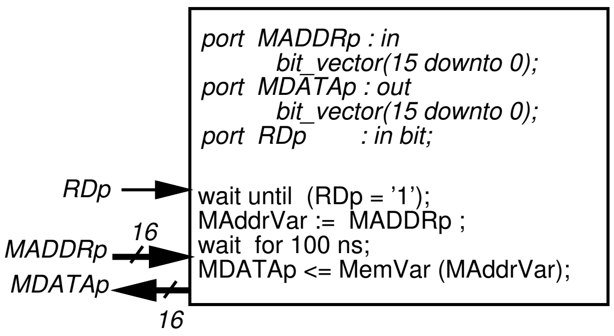
Protocol specication : HDLs

- Advantages:
  - Functionality can be veried by simulation
  - Easy to specify conditional and repetitive event sequences

- Disadvantages:
  - Cumbersome to represent timing constraints between events



Protocol Pa



Protocol Pb

Interface process generation

- Input: HDL description of two xed, but incompatible protocols
- Output: HDL process that translates one protocol to the other
  - i.e. responds to their control signals and sequence their data transfers
- Four steps required for generating interface process (IP):
  - Creating relations
  - Partitioning relations into groups
  - Generating interface process statements
  - interconnect optimization

IP generation: creating relations

- Protocol represented as an ordered set of relations
- Relations are sequences of events/actions

Protocol Pa

```
ADDRp <=  AddrVar(7 downto 0);
ARDYp <= '1';
wait until (ARCVp = '1' );
ADDRp <= AddrVar(15 downto 8);
DREQp <= '1';
wait  until (DRDYp = '1');
DataVar <=  DATAp;
```

→

Relations

```
A1 [ (true)   :
        ADDRp <=  AddrVar(7 downto 0)
        ARDYp <= '1'  ]
A2 [ (ARCVp = '1')  :
        ADDRp <=  AddrVar(15 downto 8)
        DREQp <= '1'  ]
A3 [ (DRDYp = '1')  :
        DataVar <=  DATAp  ]
```

IP generation: partitioning relations

- Partition the set of relations from both protocols into groups.
- Group represents a unit of data transfer

Protocol Pa		**Protocol Pb**	
A1 (8 bits out) **A2** (8 bits out)	→	**B1** (16 bits in)	*G1*
A3 (16 bits in)	←	**B2** (16 bits out)	*G2*

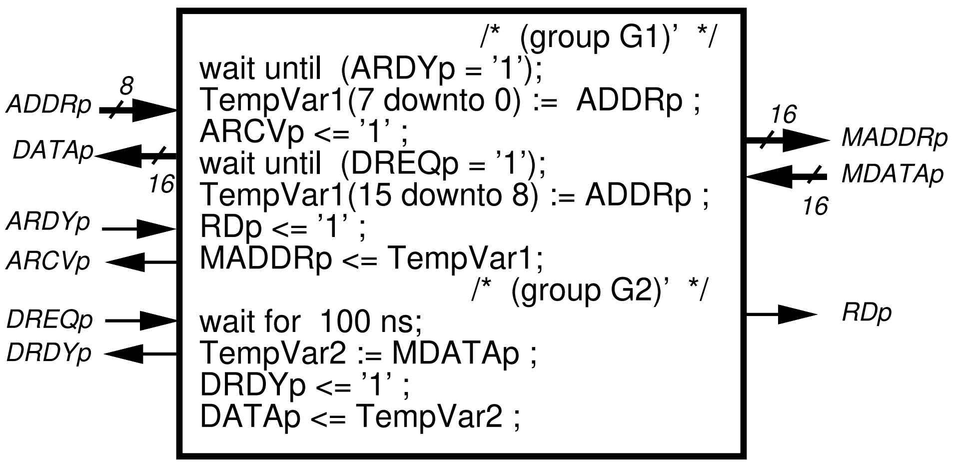
$$G_1 = (\ A_1\ A_2\ B_1\ ) \qquad G_2 = (\ B_1\ A_3\ )$$

IP generation: inverting protocol operations

- For each operation in a group, add its *dual* to interface process
- Dual of an operation represents the complementary operation
- Temporary variable may be required to hold data values

Atomic operation	*Dual operation*
wait until (Cp = '1')	Cp <= '1'
Cp <= '1'	wait until (Cp = '1')
var <= Dp	Dp <= TempVar
Dp <= var	TempVar := Dp
wait for 100 ns	wait for 100 ns

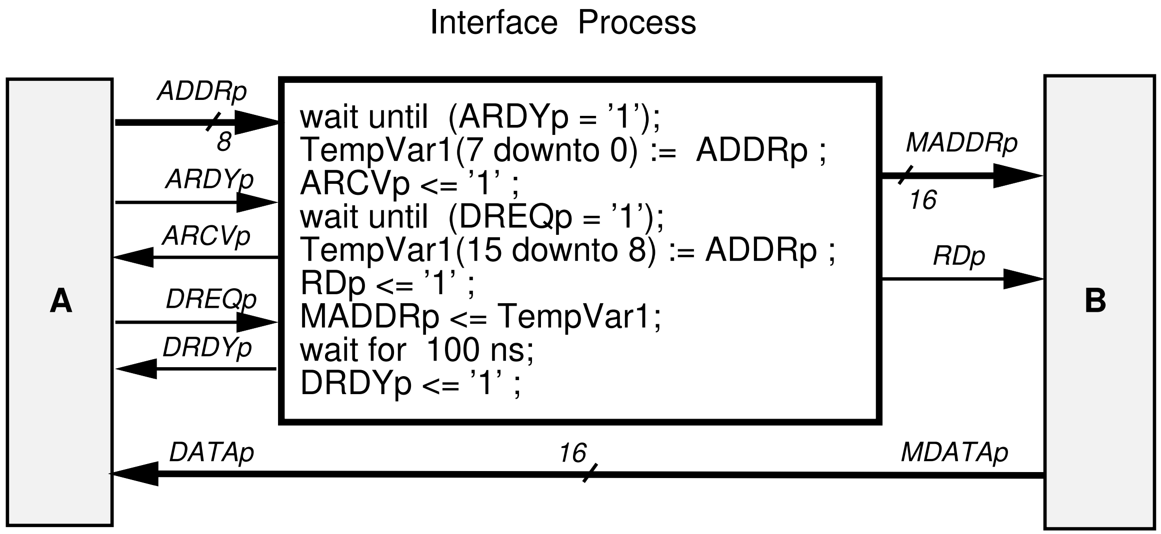
Interface Process



UC Irvine

IP generation: interconnect optimization

- Certain ports of both protocols may be directly connected
- Advantages:
  - Bypassing interface process reduces interconnect cost
  - Operations related to these ports can be eliminated from interface process



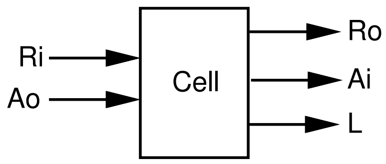
UC Irvine

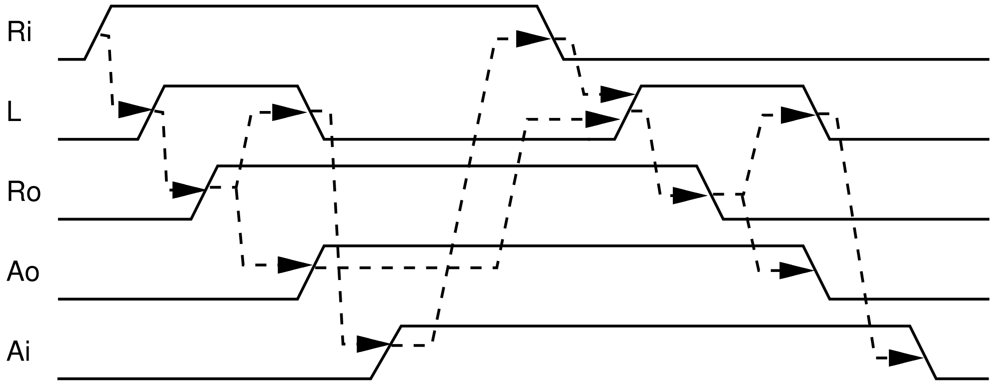
Transducer synthesis [BK87]

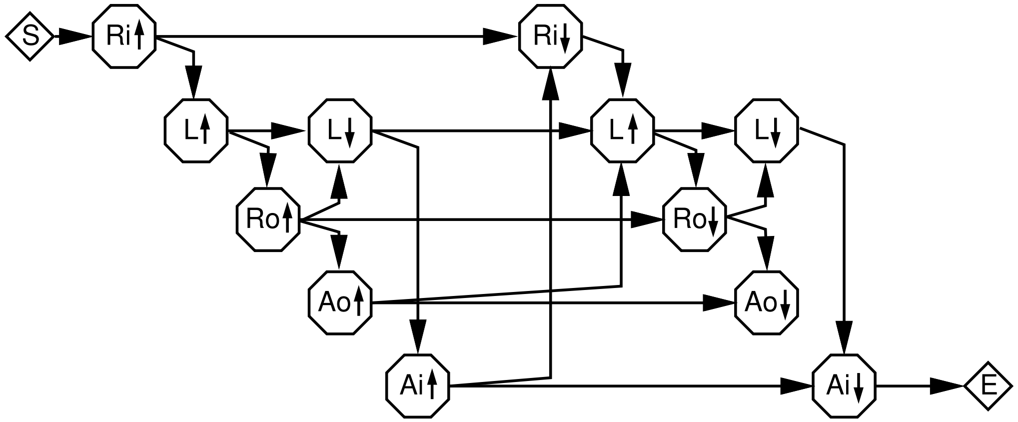
- Input: Timing diagram description of two xed protocols
- Output: Logic circuit description of transducer
- Steps for generating logic circuit from timing diagrams:
  - Create event graphs for both protocols
  - Connect graphs based on data dependencies or explicitly specied ordering
  - Add templates for each output node in combined graph
  - Merge and connect templates
  - Satisfy min/max timing constraints
  - Optimize skeletal circuit

Generating event graphs from timing diagrams

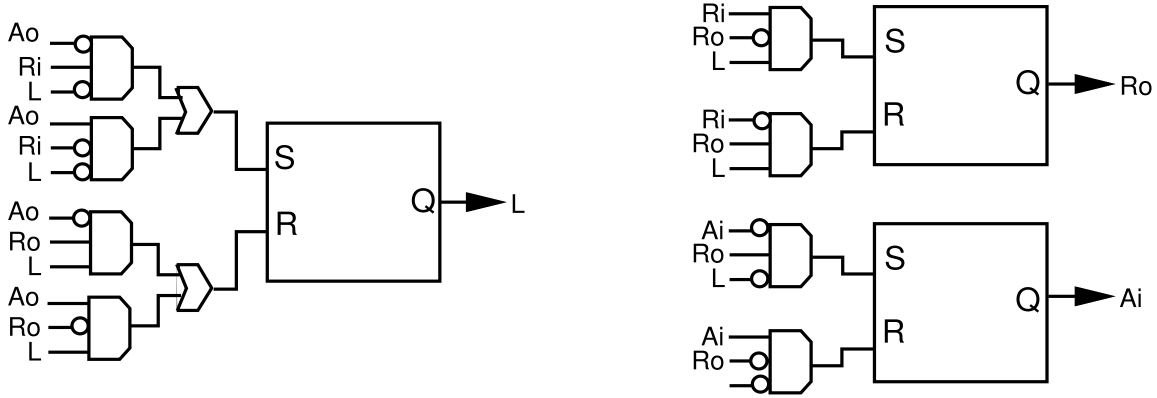
e.g. FIFO stack control cell







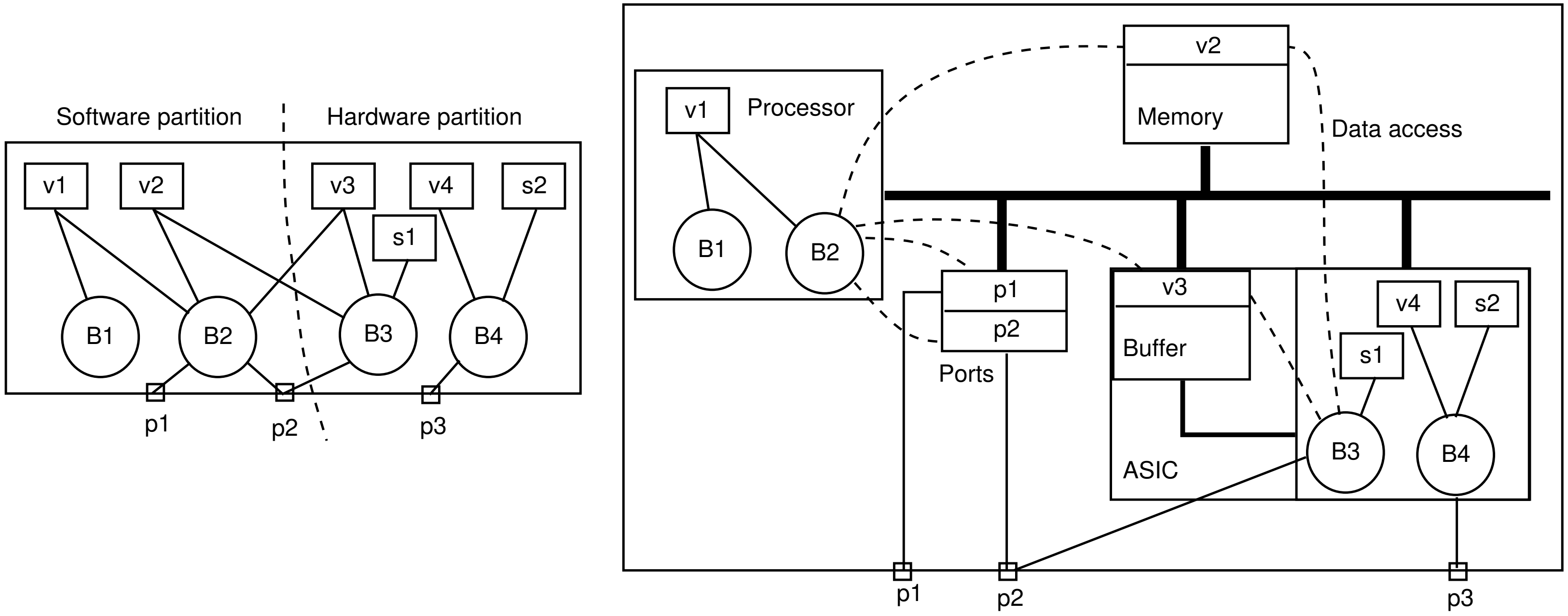
Deriving skeletal circuit from event graph



- Advantages:
  - Synthesizes logic for transducer circuit directly
  - Accounts for min/max timing constraints between events

- Disadvantages:
  - Cannot interface protocols with different data port sizes
  - Transducer not simulatable with timing diagram description of protocols

Hardware/Software interface renement



(a) Partitioned specification

(b) Mapping to architecture

Tasks of hardware/software interfacing

- Data access (e.g., behavior accessing variable) renement
- Control access (e.g., behavior starting behavior) renement
- Select bus to satisfy data transfer rate and reduce interfacing cost
- Interface software/hardware components to standard buses
- Schedule software behaviors to satisfy data input/output rate
- Distribute variables to reduce ASIC cost and satisfy performance

Summary and future directions

- In this section, we described:
  - Renement of variable groups: variable folding, address translation
  - Renement of channel groups: bus and protocol generation
  - Resolution of access conicts: arbiter generation
  - Renement of incompatible interfaces: IP generation, transducer synthesis
- Future work should address the following issues:
  - Effects of bus arbitration delays on performance of a behavior
  - Developing metrics to guide selection of protocols and arbitration schemes
  - Efcient synthesis of arbiter and interface processes

Methodology

- Past design effort focused on lower levels
- Higher levels lack well-dened methodology and tools
- Paradigm shift to higher levels can increase productivity
- Need methodology and tools for system level

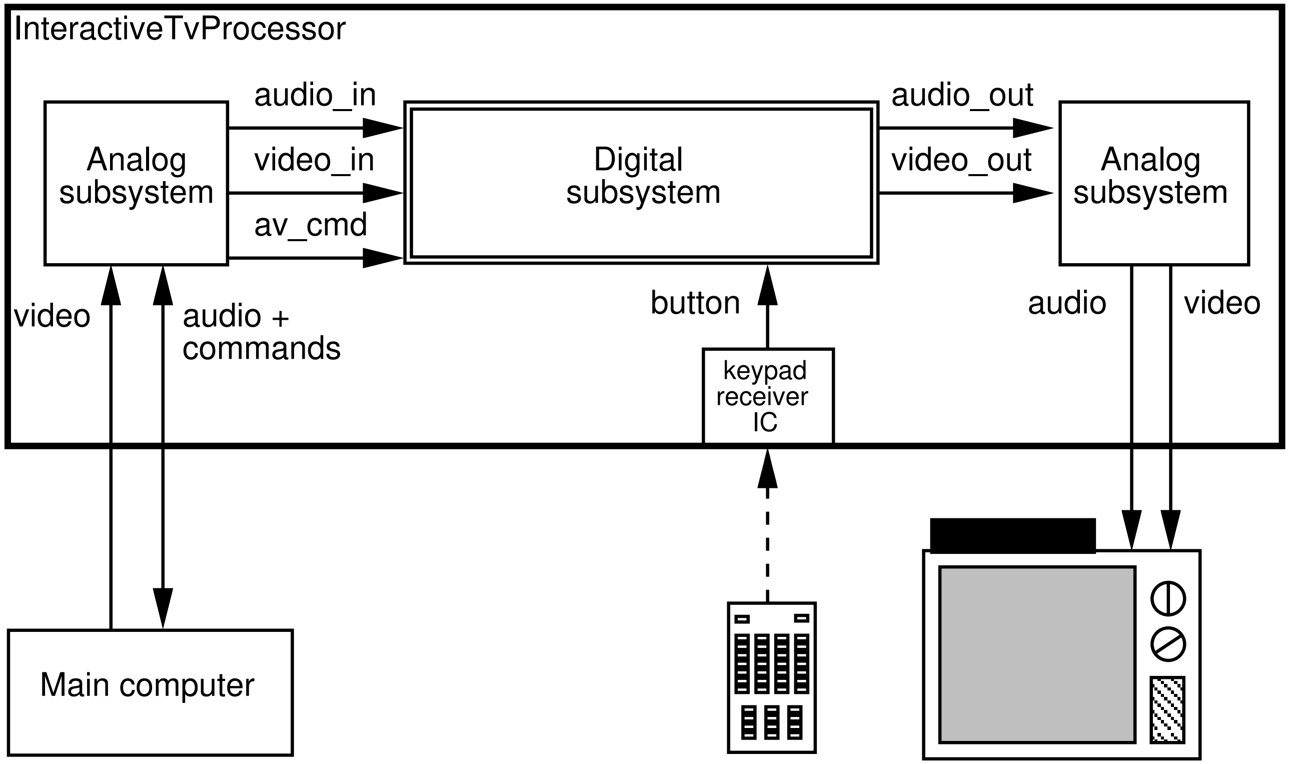
Outline

- Basic concepts in design methodology
- Example
- A design methodology
- A generic synthesis system
- Conceptualization environment

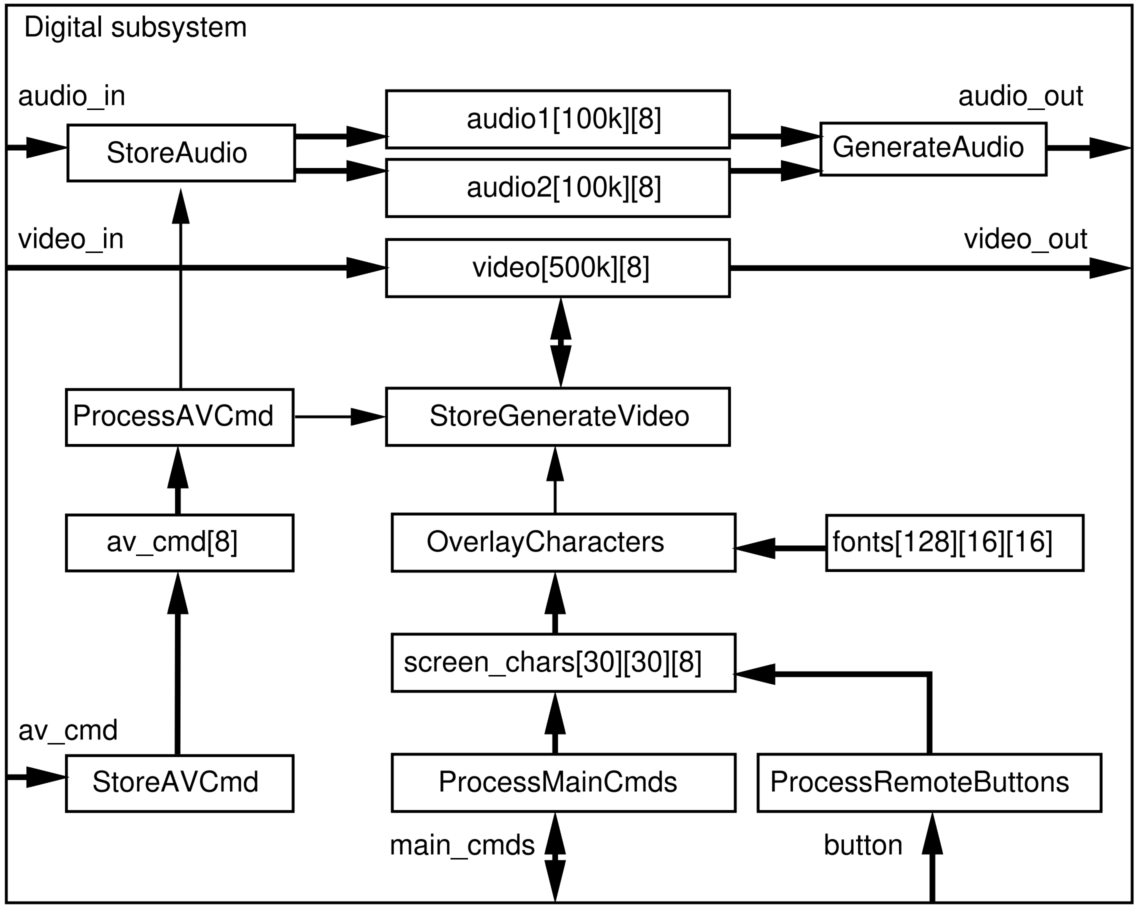
Items a design methodology must specify

- Syntax and semantics of input and output
- Algorithms for transforming input to output
- Components to be used in the design implementation
- Denition and ranges of constraints
- Mechanism for selection of architectural styles
- Control strategies (scenarios or scripts)

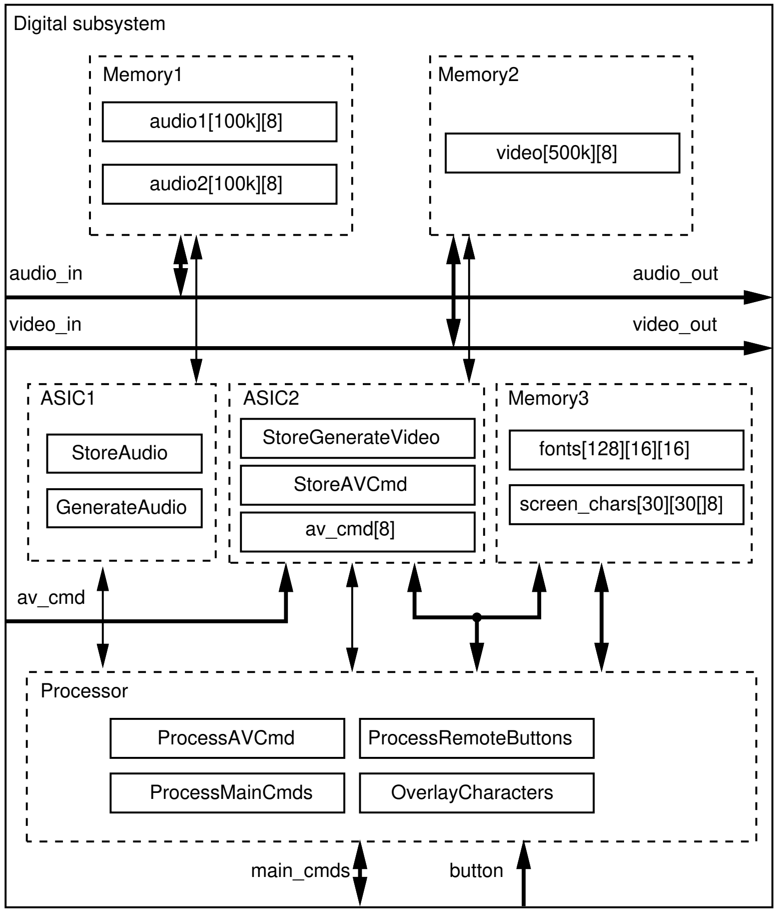
Example: Interactive TV processor



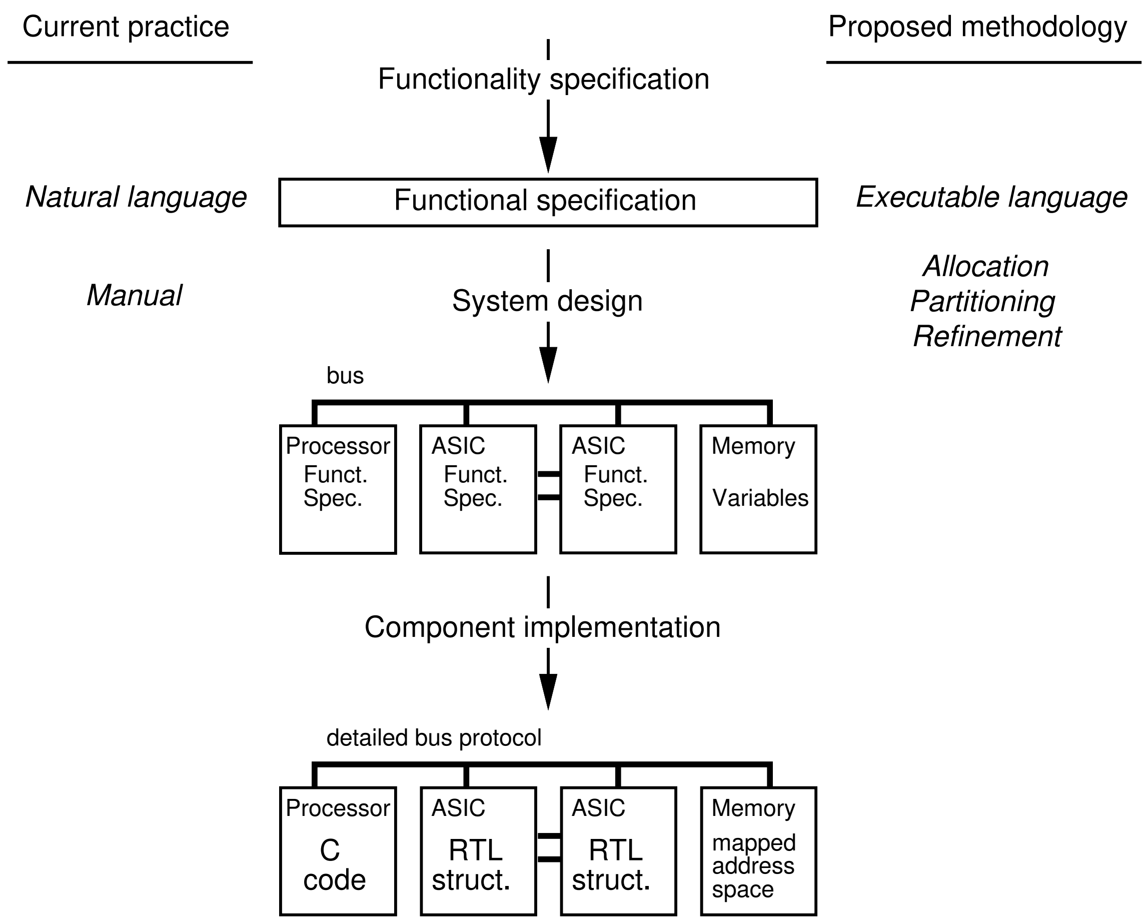
Example's dataow behavior



Example's implementation after system design



An example design methodology

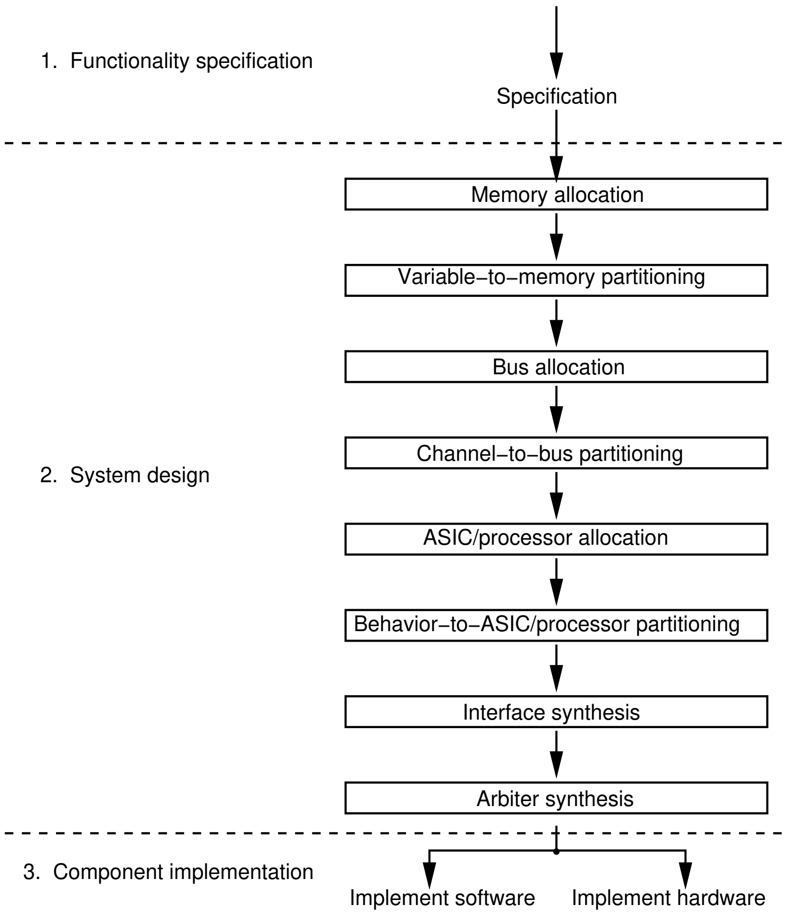


System-design tasks

System–design tasks

Functional objects	Allocation	Partitioning	Refinement
Variables	Memories	Variables to memories	Address assignment
Behaviors	Processors	Behaviors to processors	Interfacing
Channels	Buses	Channels to buses	Arbitration/protocols

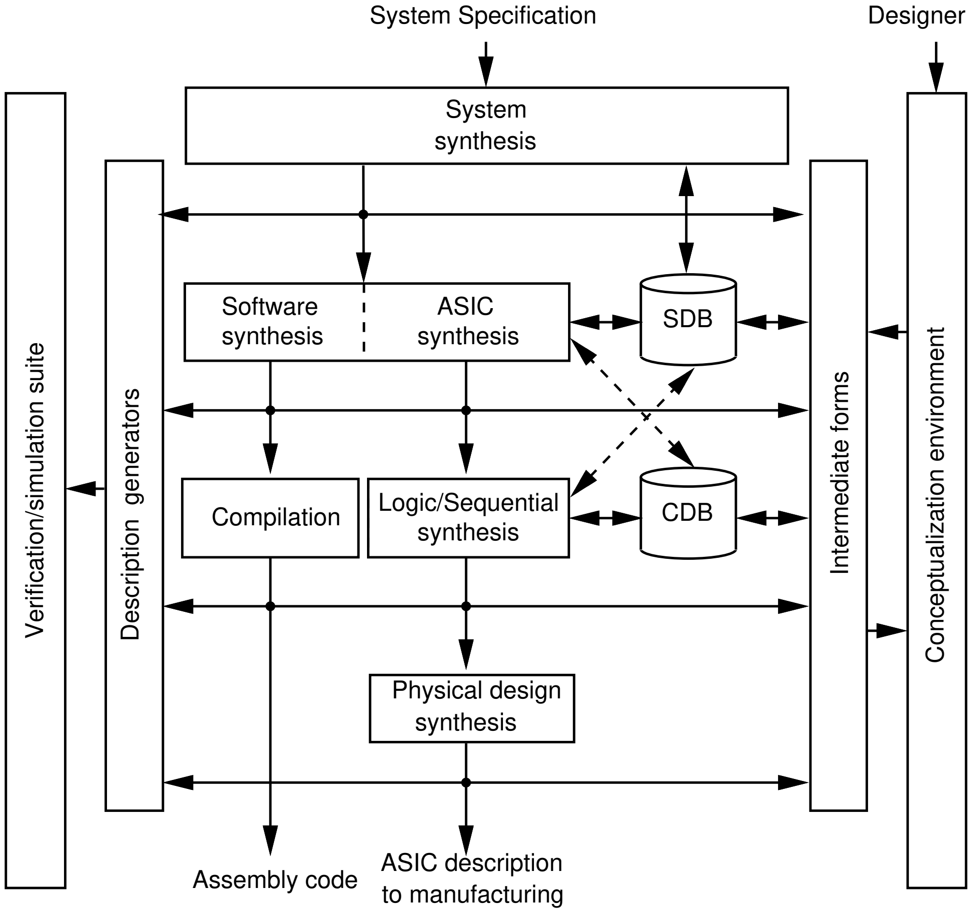
One possible ordering of tasks



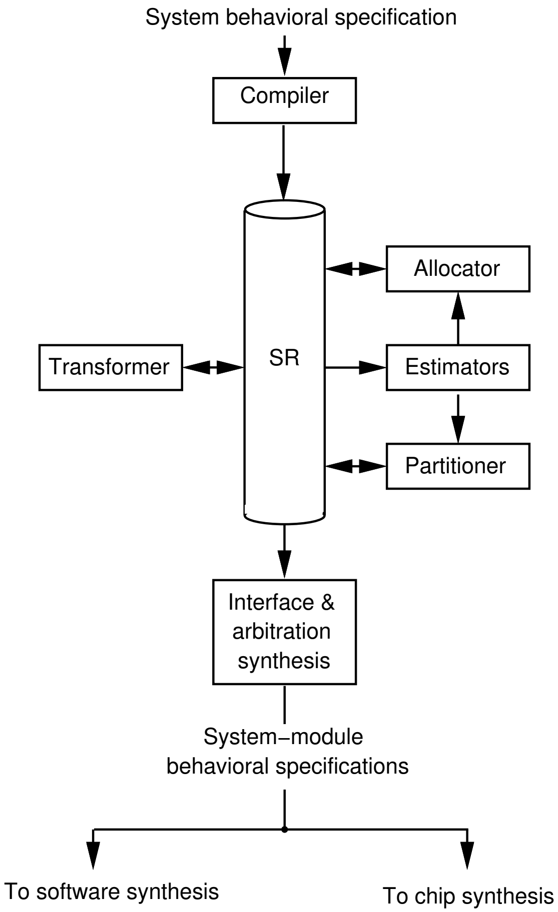
Generic synthesis system requirements

- Completeness
  - All levels of design, all implementation styles
- Extensibility
  - Allow addition of new algorithms and tools
- Controllability
  - User control of tools, design-quality feedback
- Interactivity
  - Partial design, design modication
- Upgradability
  - Evolve to describe-and-synthesize method

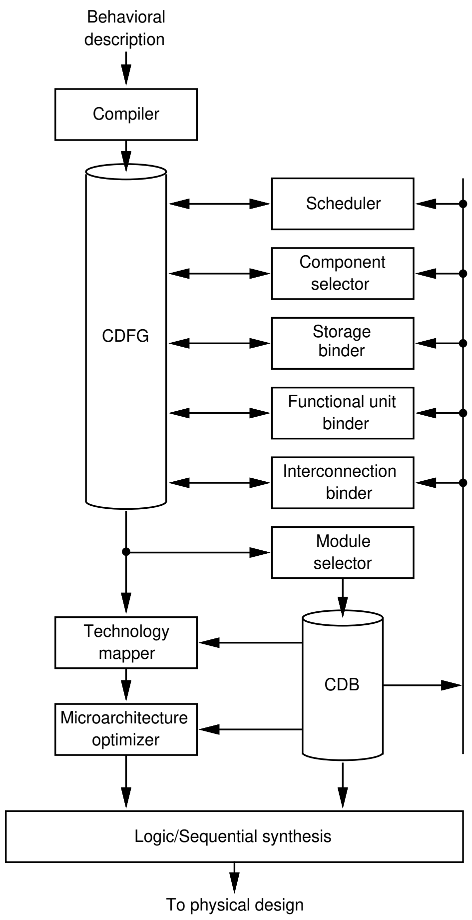
A generic synthesis system



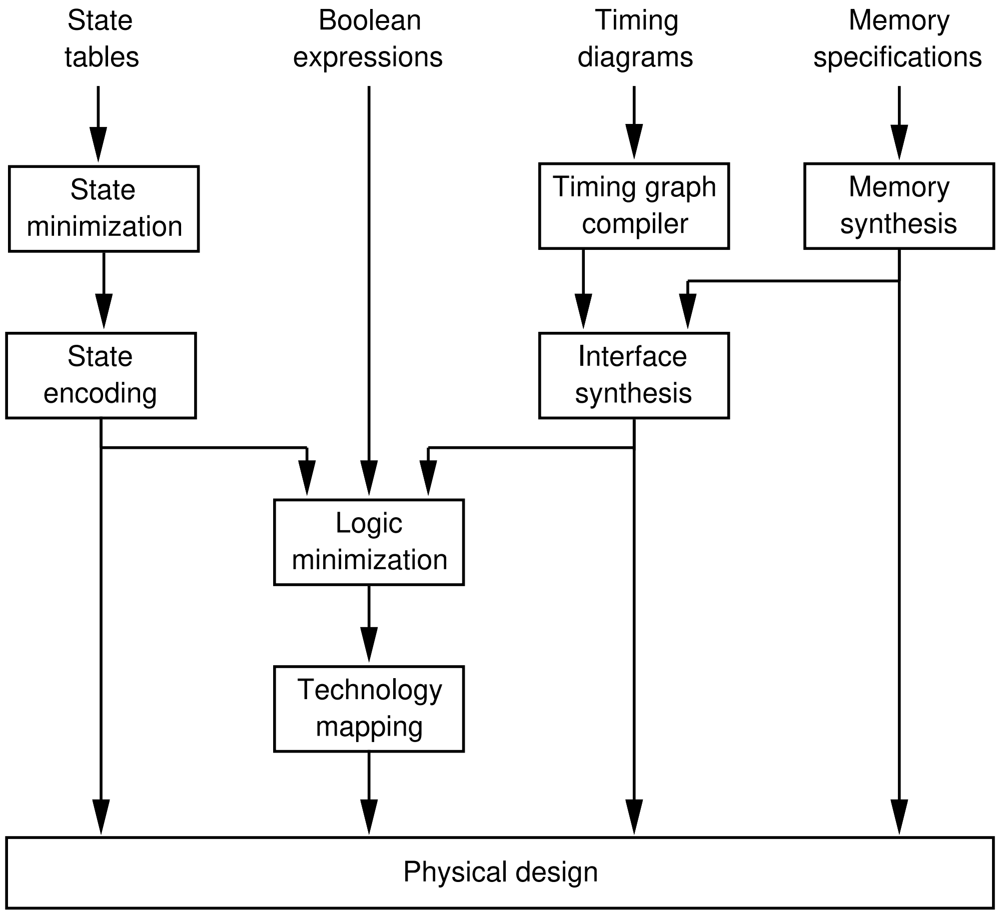
A generic system-synthesis tool



A generic chip-synthesis tool



A generic logic-synthesis tool



Conceptualization environment

- Tool is only effective if the designer can use it
  - Understandable display of data
  - Highlight design parts that need attention

- Must support many design avenues

A system-synthesis tool interface

- Allocation
- Partition
- Estimates
- Constraints

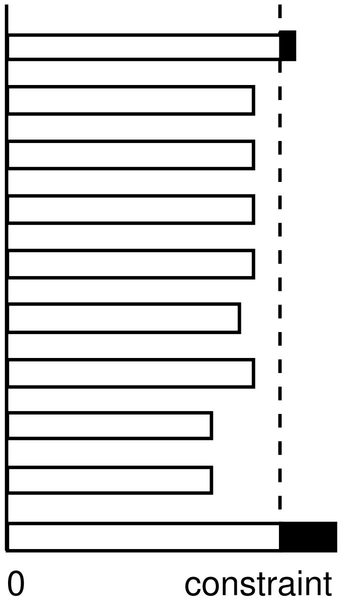
Mappings	Module type	$	Execution time	Area	Pins	Instr
System		105 /100*				
ASIC1	X100	30		16000 /20000	46/60	
CaptureAudio			100/110			
GenerateAudio			100/110			
ASIC2	X100	30		18000 /20000	48/60	
CaptureGenerateVideo			100/110			
CaptureAVCmd			100/110			
Memory1	V1000	10				
audio_array1						
audio_array2						
Memory2	V1000	10				
video_array						
Processor1	Y900	25				6000 /5000*
ProcessRemoteButtons						
ProcessMiscCmds						

Cost: 5.43 | View options | Partition/Allocate | Refine

UC Irvine

An optional design view

Quality metric	Estimate/ Constraint	Violation?
$(System)	105/100	
Execution–time(CaptureAudio)	100/110	
Execution–time(GenerateAudio)	100/110	
Execution–time(CaptureGenerateVideo)	100/110	
Execution–time(CaptureAVCmd)	100/110	
Area(ASIC1)	16000/20000	
Area(ASIC2)	18000/20000	
Pins(ASIC1)	56/60	
Pins(ASIC2)	58/60	
Instr(Processor1)	6000/5000	



Summary

- Three-step design methodology
  - Functionality specication
  - System design
  - Component implementation
- Major tasks in system design
  - Allocation
  - Partitioning
  - Renement
- Generic synthesis tool
- Conceptualization environment
  - Crucial to practical use

Future directions

- Advanced estimation methods
- Formal verication
- Testability
- Frameworks and databases
- Regularity exploiting
- System-level transformations
- Feedback incorporation

References

[BHS91] F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol Specications*. Prentice Hall, 1991.

[BK87] G. Borriello and R.H. Katz. \Synthesis and optimization of interface transducer logic,". In *Proceedings of the International Conference on Computer-Aided Design*, 1987.

[CS86] C.Tseng and D.P. Siewiorek. \Automated synthesis of datapaths in digital systems,". *IEEE Transactions on Computer-Aided Design*, pages 379{ 395, July 1986.

[EHB94] R. Ernst, J. Henkel, and T. Benner. \Hardware-software cosynthesis for microcontrollers,". In *IEEE Design & Test of Computers*, pages 64{ 75, December 1994.

[FM82] C.M. Fiduccia and R.M. Mattheyses. \A linear-time heuristic for improving network partitions,". In *Proceedings of the Design Automation Conference*, 1982.

[GD90] R. Gupta and G. DeMicheli. \Partitioning of functional models of synchronous digital systems,". In *Proceedings of the International Conference on Computer-Aided Design*, pages 216{ 219, 1990.

[GD92] R. Gupta and G. DeMicheli. \System-level synthesis using re-programmable components,". In *Proceedings of the European Conference on Design Automation (EDAC)*, pages 2{ 7, 1992.

[GD93] R. Gupta and G. DeMicheli. \Hardware-software cosynthesis for digital systems,". In *IEEE Design & Test of Computers*, pages 29{ 41, October 1993.

[GVN94] D.D. Gajski, F. Vahid, and S. Narayan. \A system-design methodology: Executable-specication renement,". In *Proceedings of the European Conference on Design Automation (EDAC)*, 1994.

[Hal93] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[Hoa78] C.A.R. Hoare. \Communicating sequential processes,". *Communications of the ACM*, 21(8): 666{ 677, 1978.

[IEE88] IEEE Inc., N.Y. *IEEE Standard VHDL Language Reference Manual*, 1988.

[JMP88] R. Jain, M. Mlinar, and A. Parker. \Area-time model for synthesis of non-pipelined designs,". In *Proceedings of the International Conference on Computer-Aided Design*, 1988.

[Joh67] S.C. Johnson. \Hierarchical clustering schemes,". *Psychometrika*, pages 241{ 254, September 1967.

[KC91] Y.C. Kirkpatrick and C.K. Cheng. \Ratio cut partitioning for hierarchical designs,". *IEEE Transactions on Computer-Aided Design*, 10(7): 911{ 921, 1991.

[KGV83] S. Kirkpatrick, C.D. Gelatt, and M. P. Vecchi. \Optimization by simulated annealing,". *Science*, 220(4598): 671{ 680, 1983.

[KL70] B.W. Kernighan and S. Lin. \An efcient heuristic procedure for partitioning graphs,". *Bell System Technical Journal*, February 1970.

[LT91] E.D. Lagnese and D.E. Thomas. \Architectural partitioning for system level synthesis of integrated circuits,". *IEEE Transactions on Computer-Aided Design*, July 1991.

[MK90] M.C. McFarland and T.J. Kowalski. \Incorporating bottom-up design into hardware synthesis,". *IEEE Transactions on Computer-Aided Design*, September 1990.

[NG92] S. Narayan and D.D. Gajski. \System clock estimation based on clock slack minimization,". In *Proceedings of the European Design Automation Conference (EuroDAC)*, 1992.

[NG94] S. Narayan and D.D. Gajski. \Synthesis of system-level bus interfaces,". In *Proceedings of the European Conference on Design Automation (EDAC)*, 1994.

[NVG92] S. Narayan, F. Vahid, and D.D. Gajski. \System specication with the SpecCharts language,". In *IEEE Design & Test of Computers*, Dec. 1992.

[PK89] P.G. Paulin and J.P. Knight. \Algorithms for high-level synthesis,". In *IEEE Design & Test of Computers*, Dec. 1989.

[PPM86] A.C. Parker, T. Pizzaro, and M. Mlinar. \MAHA: A program for datapath synthesis,". In *Proceedings of the Design Automation Conference*, 1986.

[TM91] D.E. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1991.

[VG92] F. Vahid and D.D. Gajski. \Specication partitioning for system design,". In *Proceedings of the Design Automation Conference*, 1992.

[VGG93] F. Vahid, J. Gong, and D.D. Gajski. \A hardware-software partitioning algorithm for minimizing hardware,". UC Irvine, Dept. of ICS, Technical Report 93-38,1993.