ASYNC 2004
19-23 April
Hersonissos - Crete

Compiling C to Asynchronous Hardware

Mihai Budiu, Girish Venkataramani,
Tiberiu Chelcea, Seth Goldstein

{mihaib, girish, tibi, seth}@cs.cmu.edu

Carnegie Mellon

www.cs.cmu.edu/~phoenix

Outline

- Context: Future of Electronics
- Overview of Compilation Process
- From C to Dataflow
- From Dataflow to Asynchronous Circuits
- Demo [throughout the presentation]

Moore's Law





Moore's Law

Imagine: Computers that

-
-
-
-

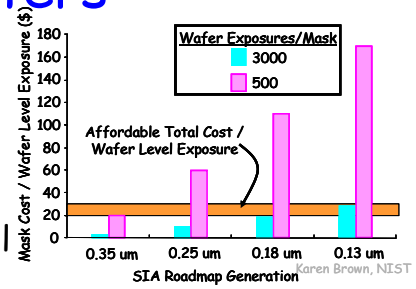
Imagining it is hard enough, achieving it requires a rethink of the entire tool chain.

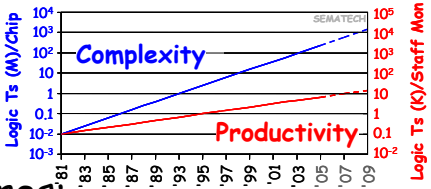
Computers from atomic scale components


Size Matters

As we scale down:

- Devices become
  - More variable
  - More faulty (defects & faul
  - More numerous
- Fabrication becomes
  - More constrained
  - More expensive
- Design becomes
  - More complicated
  - More expensive





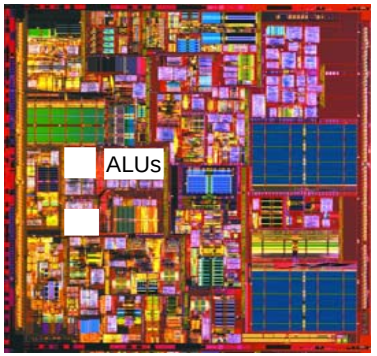
Requires:

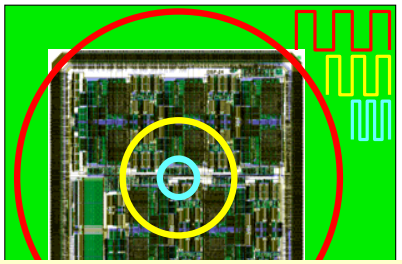
- Defect tolerant architectures
- Higher level specification
- Universal substrate

Technical Challenges

Delay

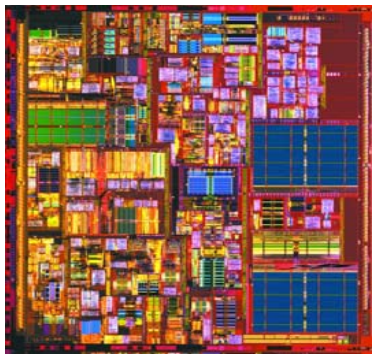
gate	5ps
wire	10ps

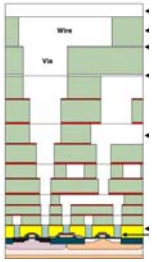


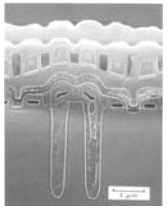


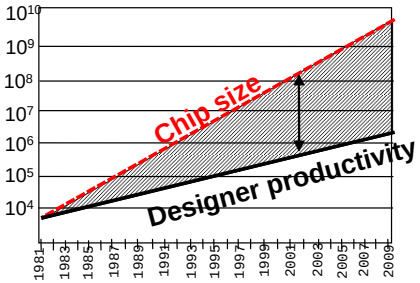
Cannot rely on global signals (clock is a global signal)

Compexity Challenges





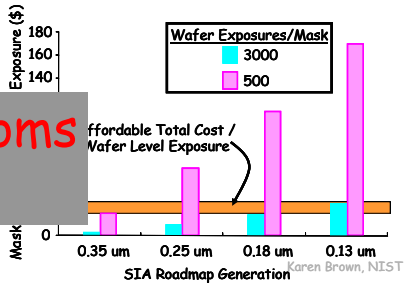




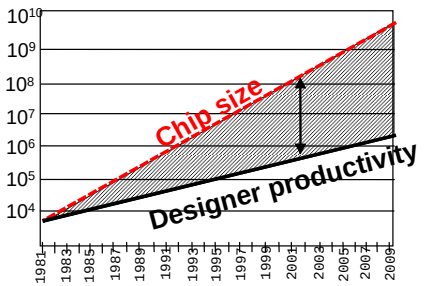
+Testing
+Verification

$$$$$$ Challenges

Mask defect of 3 atoms
→ chip defect!



+Testing
+Verification



Manufacturing Paradigm Shift Required

Today

- Reliable Systems from reliable components
- Functionality invested at time of manufacture
- Behavior remains same as features scales down

Manufacturing Paradigm Shift Required

Today

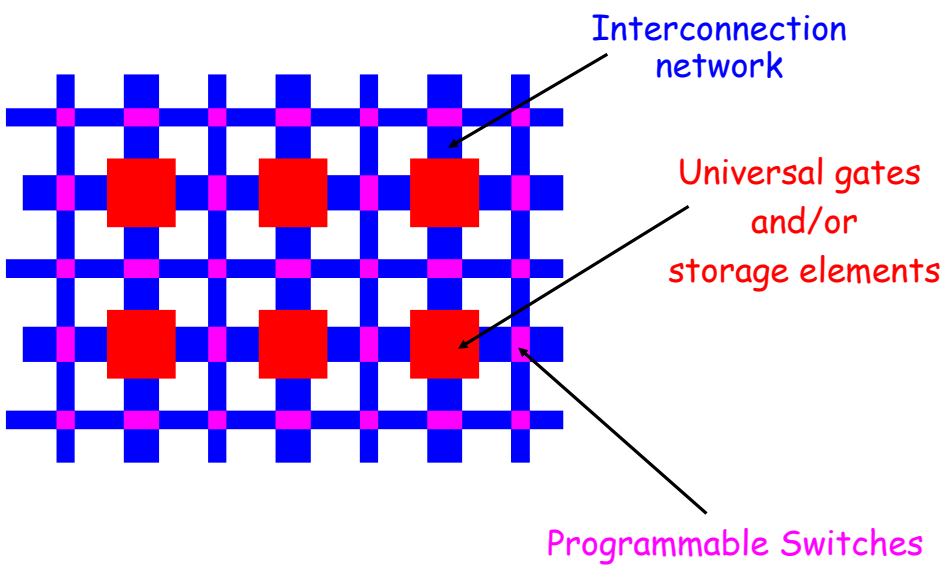
Future

- Reliable Systems from reliable components
  Reliable systems from unreliable components
- Functionality invested at time of manufacture
  Functionality modified after manufacture
  New manufacturing: Bottom-up assembly
- Behavior remains same as features scales down
  Expect increased variability
  Changes in functionality
  Restrictions on connectivity

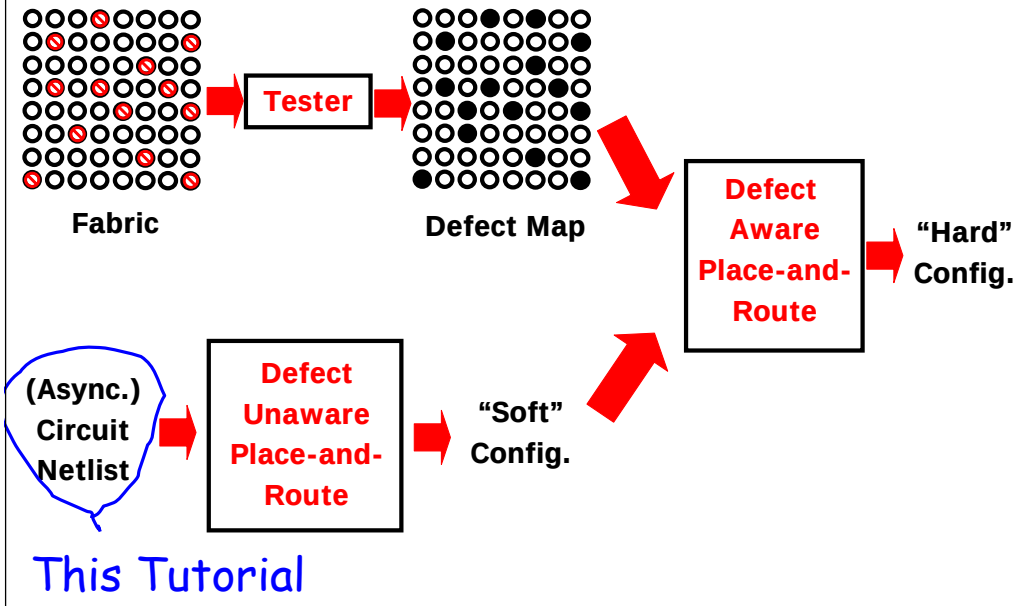
Defect Tolerant Architectures

- Features:
  - Regular topology
  - Homogenous resources
  - Fine-grained?
  - Post-fabrication modification
- Example from today: DRAM
  - Requires external device for testing
  - Requires external device for repair
- Logic? FPGA

FPGA

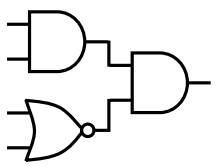


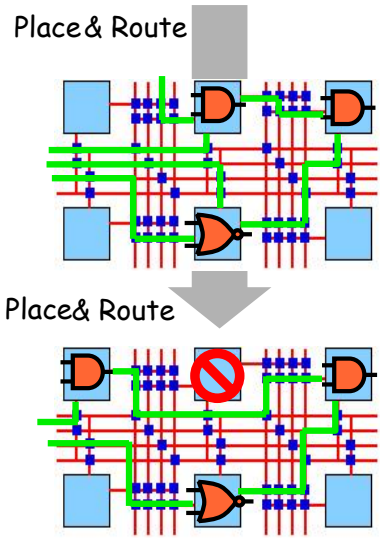
DFT tool flow



This Tutorial

Reconfigurability & DFT

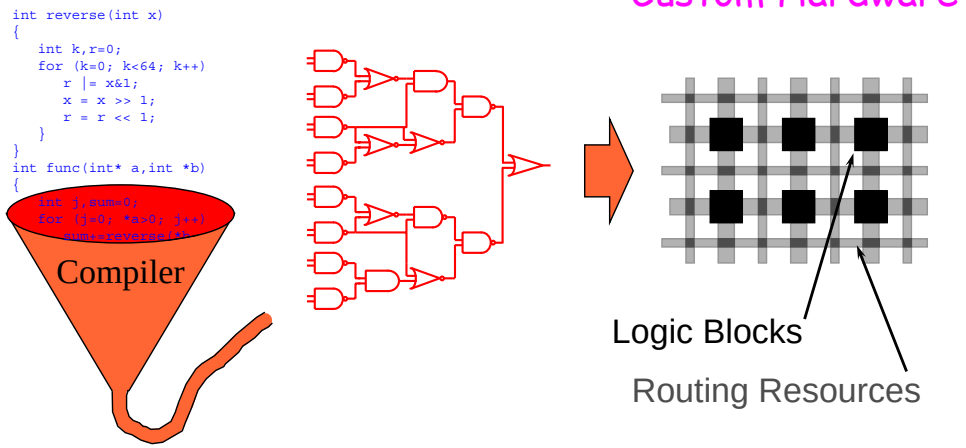




- FPGA computing fabric
  - Regular
  - periodic
  - Fine-grained
  - Homogenous
- programs ⇒ circuits
- Aides defect tolerance

Reconfigurable Computing

General-Purpose Custom Hardware General-Purpose Custom Hardware



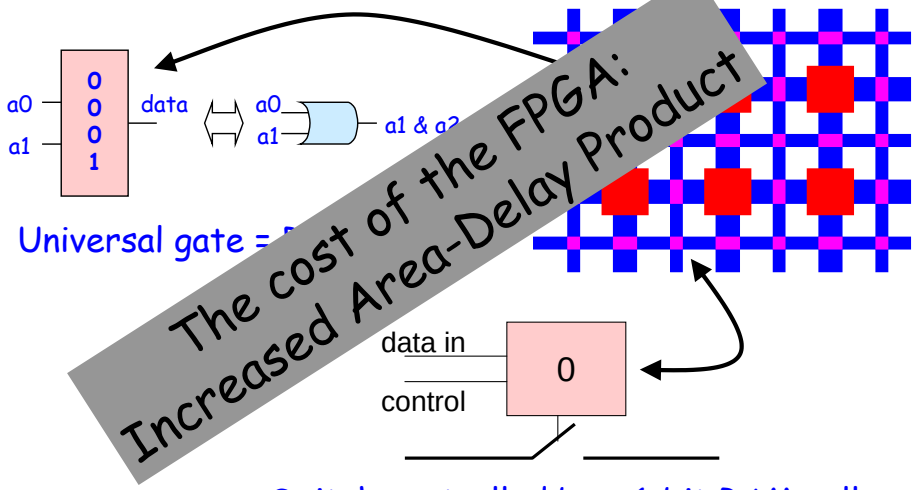
Advantages of Reconfigurable

- Flexibility of a processor
- Performance of custom hardware

Near

You have to store the configuration!

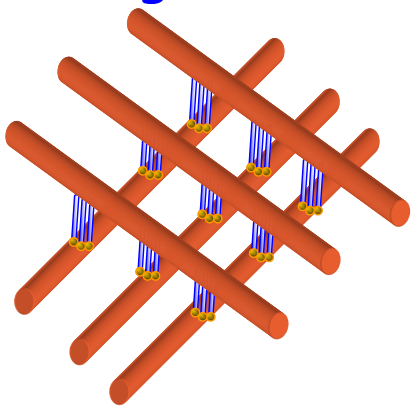
Heart of an FPGA



The cost of the FPGA: Increased Area-Delay Product

Switch controlled by a 1-bit RAM cell

The Molecular Electronics Advantage: A Reconfigurable Switch

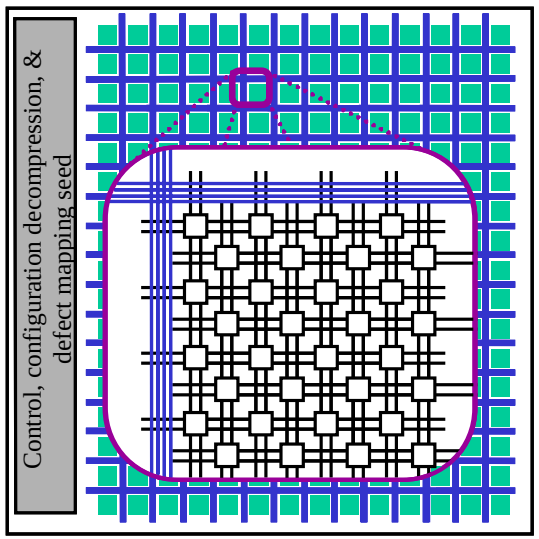


- Each crosspoint is a reconfigurable switch
- Can be programmed using the signal wires

Eliminates overhead found in CMOS FPGAs!

The NanoFabric

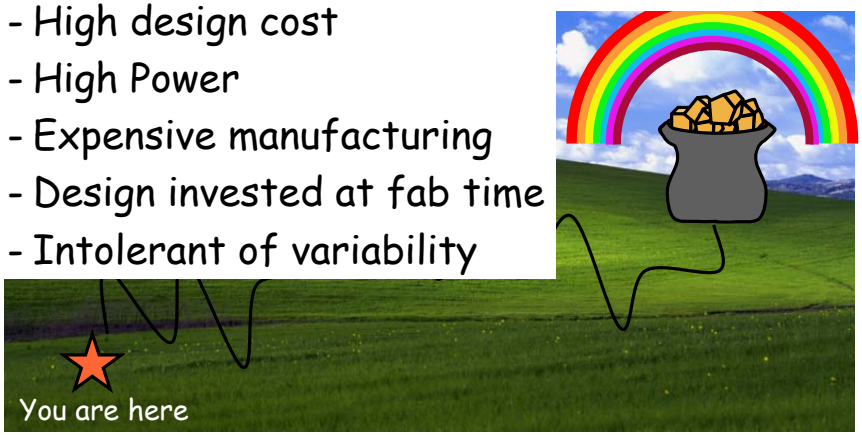
- Nanoscale layer put **deterministically** on top of CMOS
- Highly regular
- ~10^8 long lines
- ~10^6 clusters
  - Cluster has 128 blocks



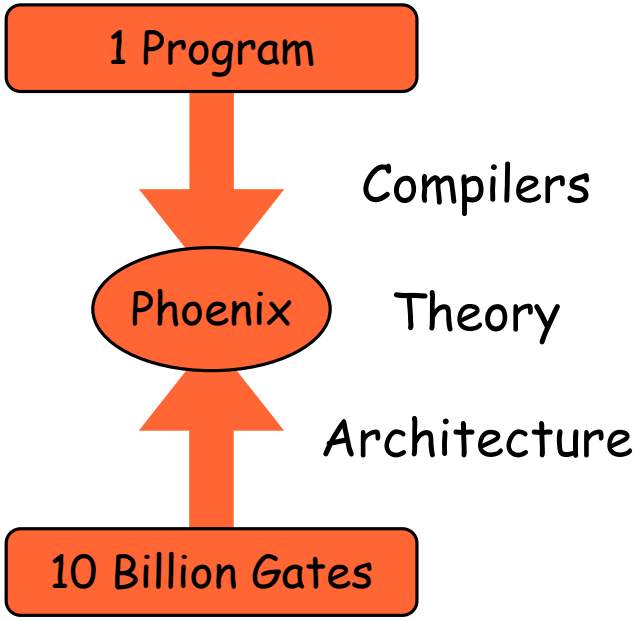
How Do W

- Low design cost
- Low power
- Inexpensive manufacturing
- Reconfigurable
- Defect and fault tolerance

- High design cost
- High Power
- Expensive manufacturing
- Design invested at fab time
- Intolerant of variability



Spanning 10-orders of Magnitude



Bryant's Law

- Processor verification is always 10 years behind
- What to do?

Don't use processors!

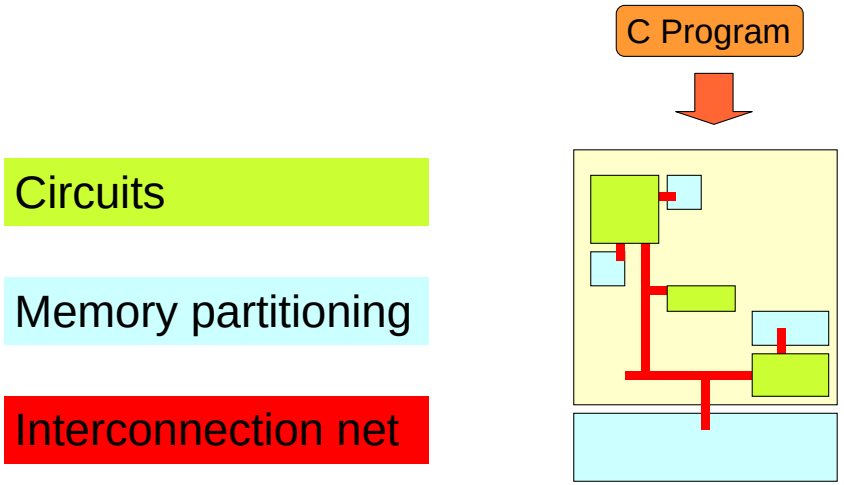
Application-Specific Hardware

Compiler-synthesized architecture

- Fast prototyping: automatic from ANSI C ⇒ HW
- High performance: sustained ≈1 GOPS [@180nm]
- Low power: Energy/op 100-1000× better than μP

From dusty-deck C program kernels

ASH: Application-Specific Hardware



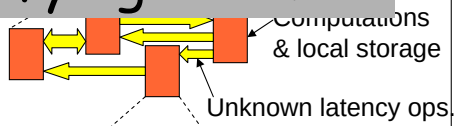
Circuits From Compilers

1. Program

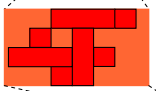
```
int reverse(int x)
{
   int k,r=0;
   for (k=0; k<64; k++)
      r |= x&1;
      x = x >> 1;
      r = r << 1;
```

Certifying Circuits!

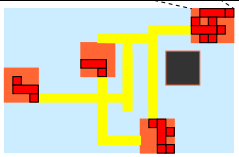
2. Split-phase Abstract Machines



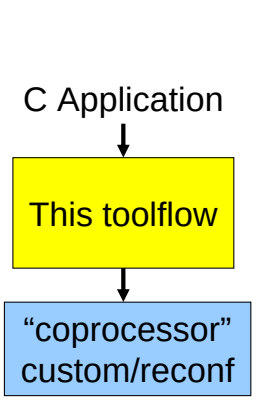
3. Configurations placed independently

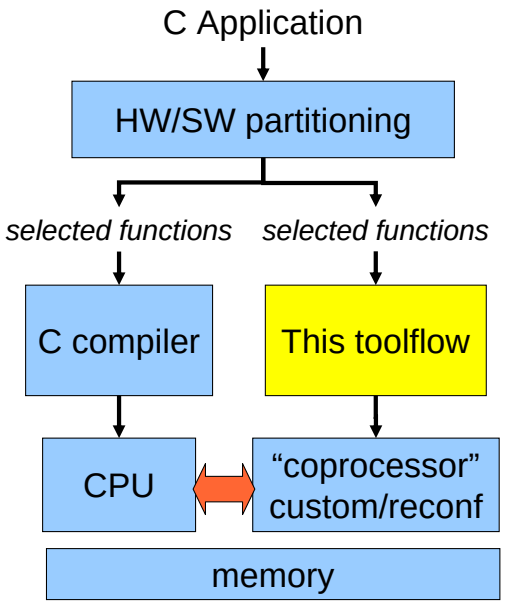


4. Placement on chip



Current Usage Methodology





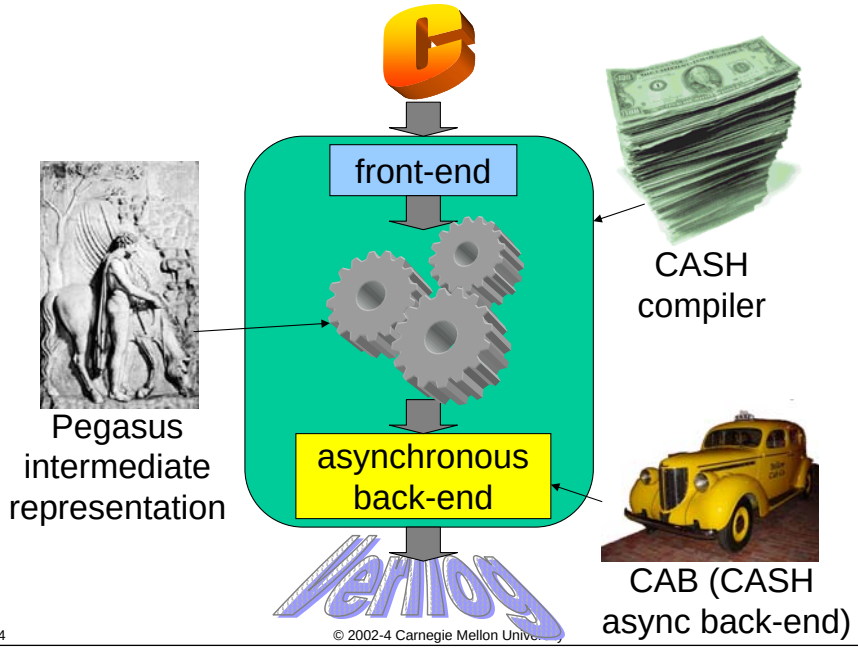
Why Asynch From C?

- Why Asynch?
  - Tolerant of variability
  - Supports defect tolerance by remapping
  - Lower power
  - Natural composability
- Why imperative sequential language?
  - Fast prototyping
  - Lower design costs
  - More human talent
  - Our first starting point:
    If we can do this, we can do anything …

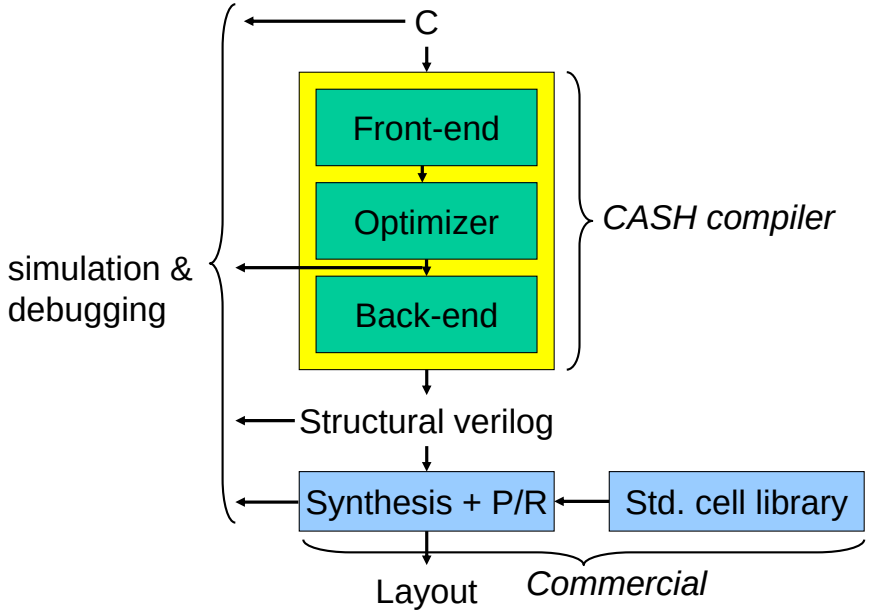
Outline

- Context: Future of Electronics
- Overview of Compilation Process
- From C to Dataflow
- From Dataflow to Asynchronous Circuits
- Demo [throughout the presentation]

CASH: Compiling for ASH



Tool-flow



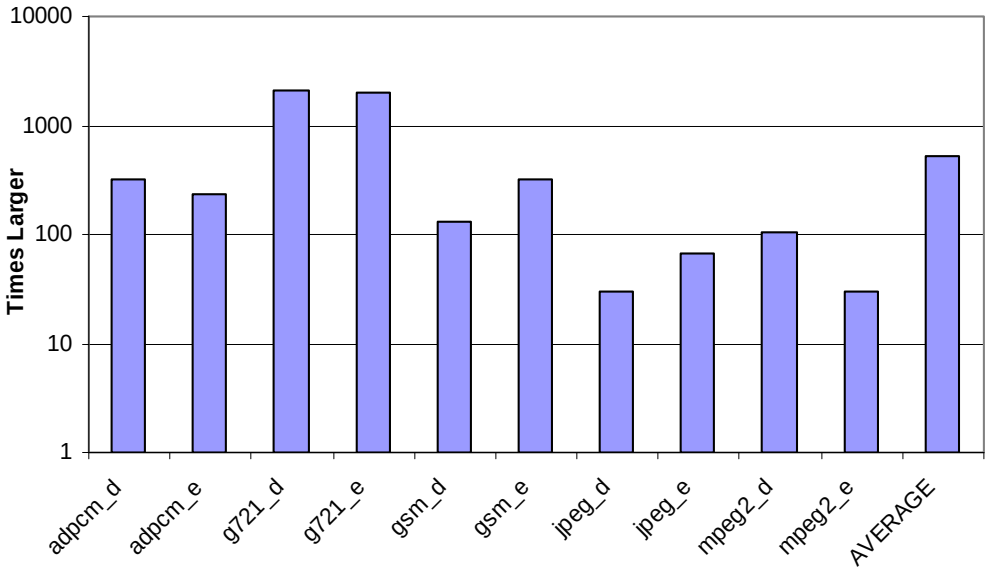
Validation

- Compiled Mediabench kernels
- Mapped to 180nm/2V library
- Fully automatic from C to layout
- Bit-accurate simulations

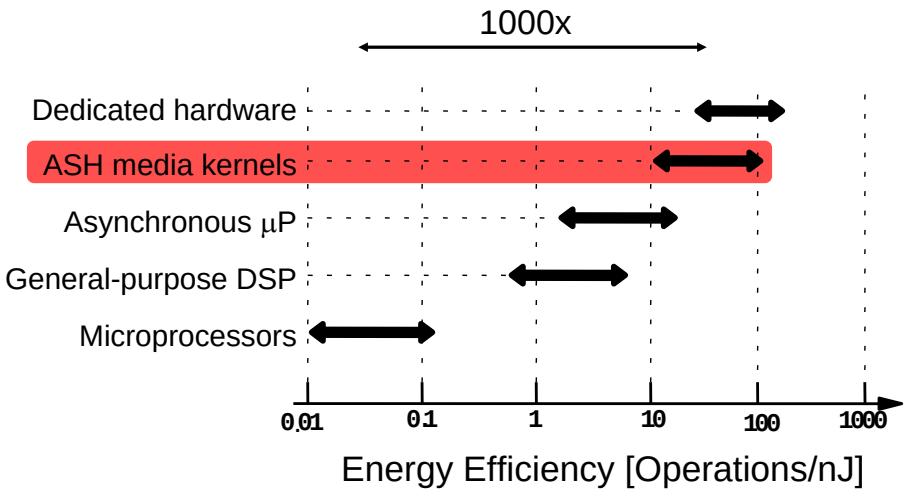
Results

- **Compile speed**: C ⇒ Verilog in seconds
- **Area**: 1-8 mm^2
- **Performance**: 500-1000 MOPS
- **Power consumption**: 5-30mW
- **Energy**: 10-160 ops/nanoJoule

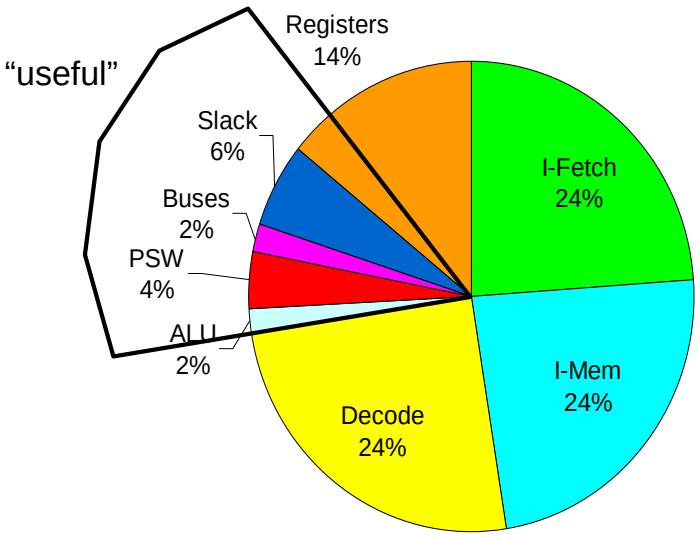
Energy-Delay (CPU/ASH)



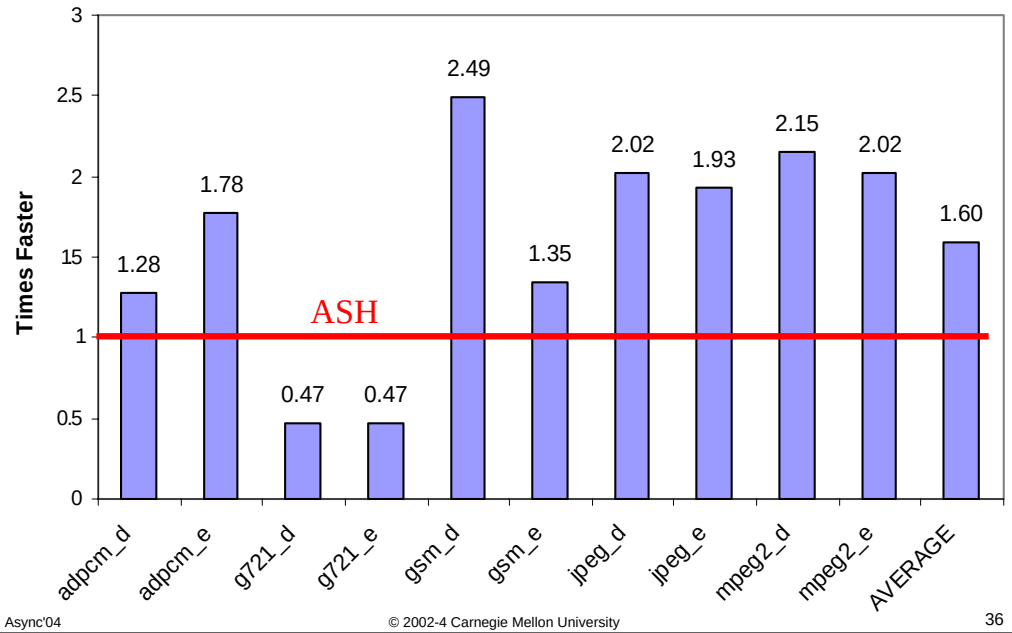
Energy Efficiency



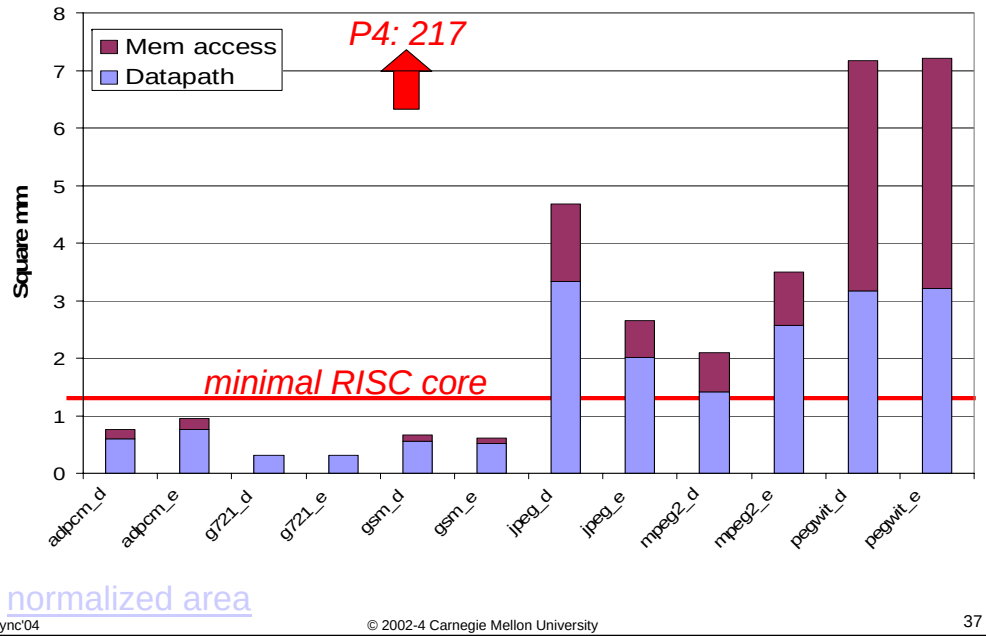
Async Processor Power



Timing: CPU times better



ASH Area



Implementation Overview

Style:

- Micropipelines
- 4-phase bundled-data handshaking

Implementation Features:

- No centralized control
- Completely parallel datapath
- Single writer for all datapath channels
- No arbitration on datapath

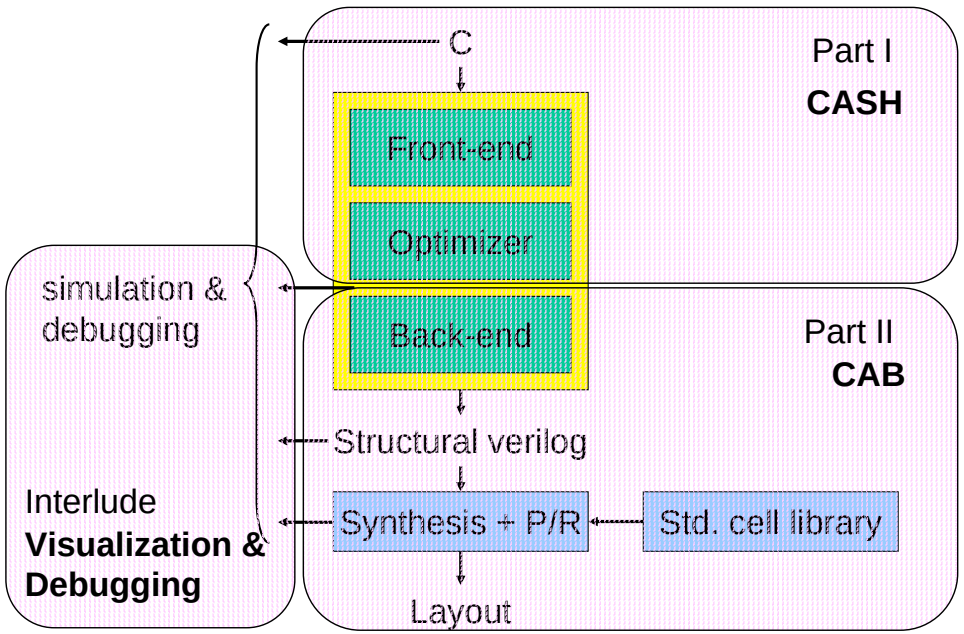
Monolithic memory:

- Pipelined arbitrated memory access net

Optimizations

- Scalar optimizations
  - unreachable/dead code, gcse, strength reduction, loop-invariant code motion, software pipelining, reassociation, algebraic simplifications, induction variable optimizations, loop unrolling, inlining
- Memory optimizations
  - dependence & alias analysis, register promotion, redundant load/store elimination, memory access pipelining, loop decoupling
- Boolean optimizations
  - Espresso CAD tool, bitwidth analysis

Tutorial Outline



Part I

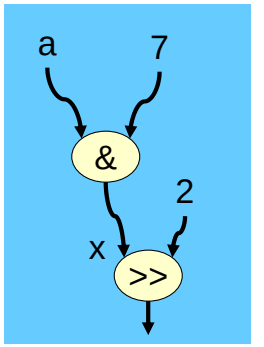
From C to Dataflow

Computation = Dataflow

Programs

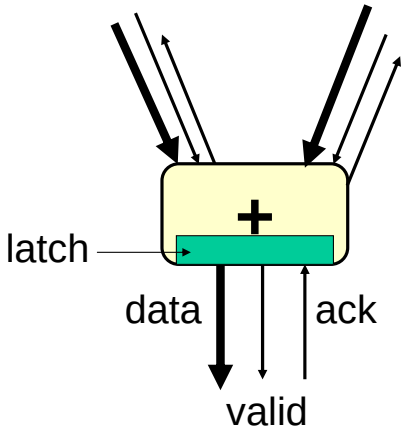
```
x = a & 7;
...
y = x >> 2;
```

Circuits

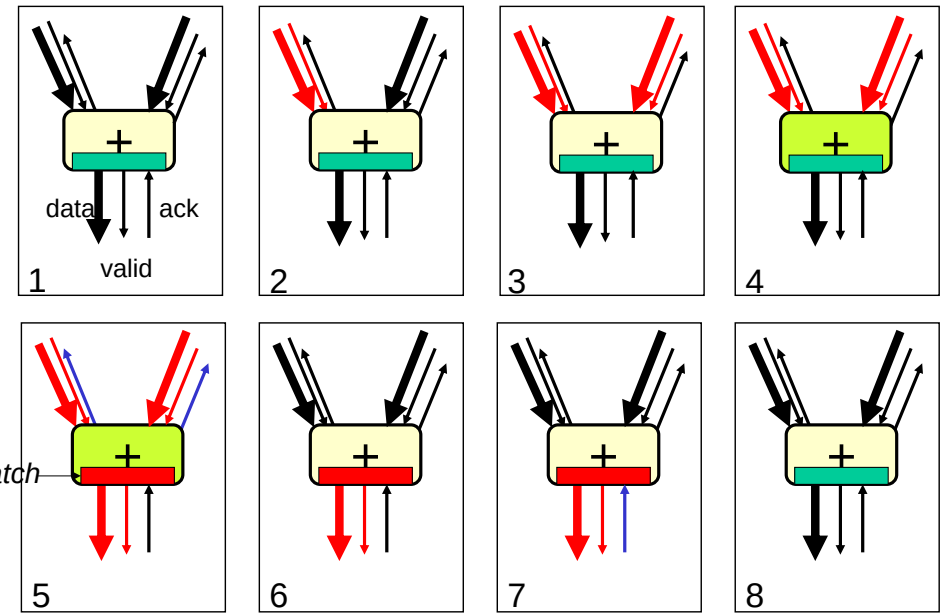


- Operations ⇒ functional units
- Variables ⇒ wires
- No interpretation

Basic Computation



Asynchronous Computation

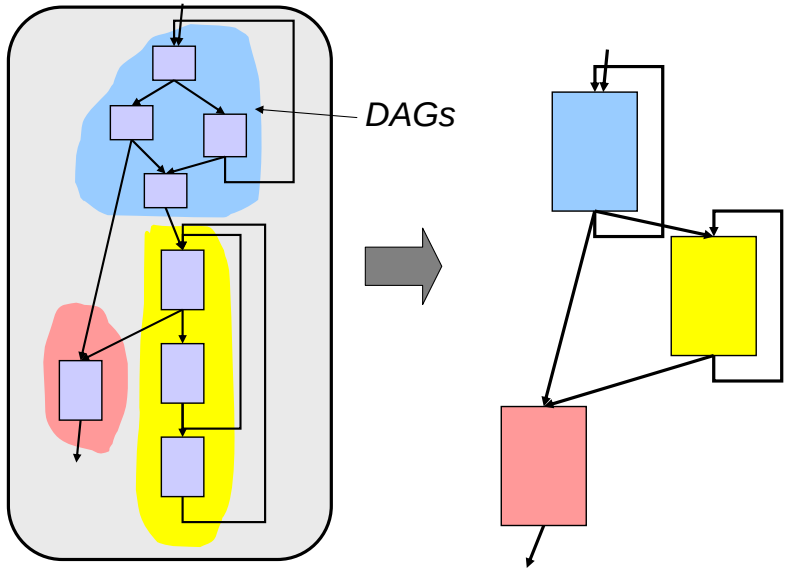


Compilation Outline

- Suif is the front-end
- standard transformation
- some optimizations

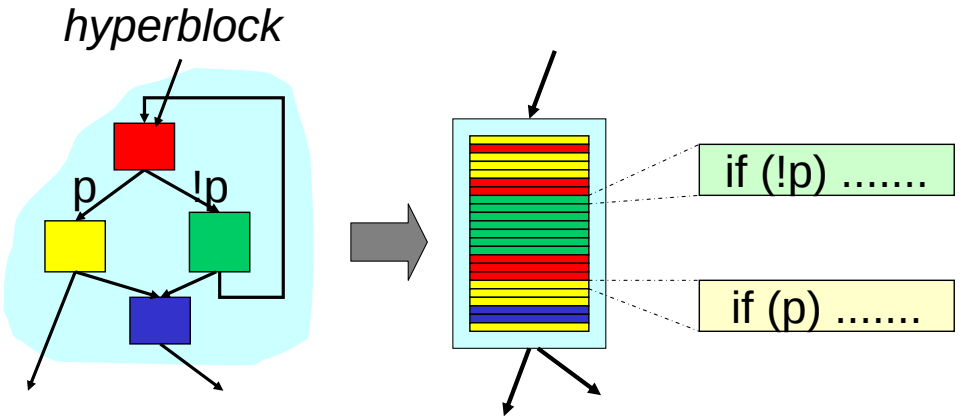
C → CFG → ∑ acyclic → dataflow → circuits

Hyperblocks



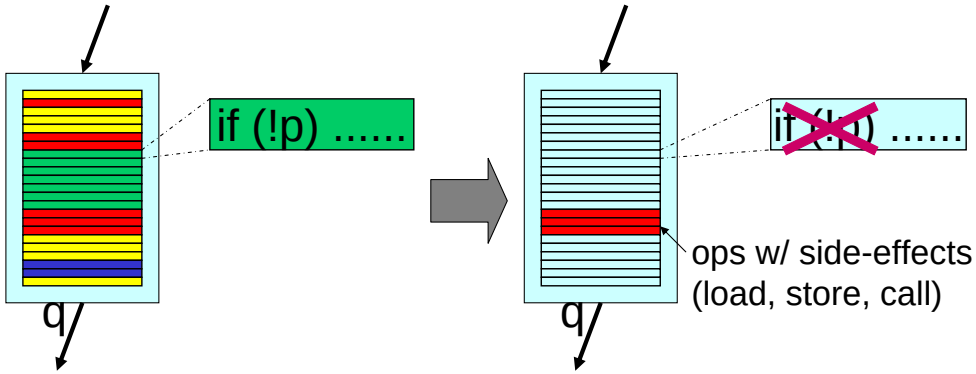
C → CFG → ∑ acyclic → dataflow → circuits

Predication



C → CFG → ∑ acyclic → dataflow → circuits

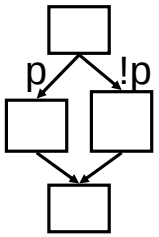
Speculation



C → CFG → ∑ acyclic → dataflow → circuits

Example Conditional

```
if (x > 0)
        y = -x;
else
        y = b*x;
... = y;
```

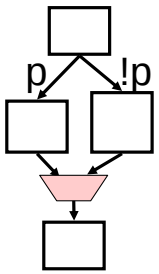


predication

```
p = (x > 0)
if (p)  y = -x;

if (!p) y = b*x;
... = y;
```

speculation

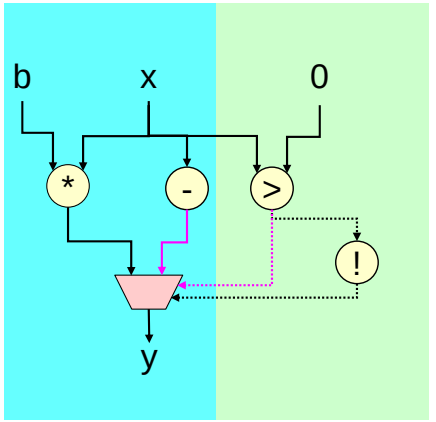
```
p = (x > 0);
y1 = -x;

y2 = b*x;
y = mux(p, !p, y1, y2);
... = y;
```



C → CFG → ∑ acyclic → dataflow → circuits

MUX = Single-Assignment

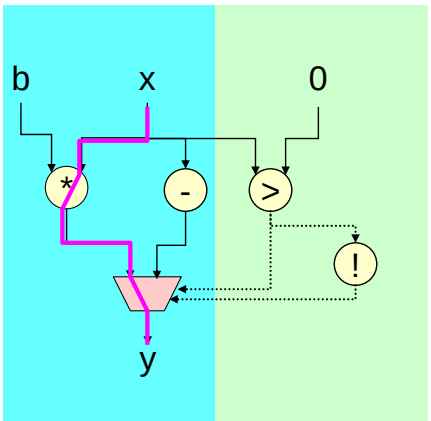
```
if (x > 0)
        y = -x;
else
        y = b*x;
```



C → CFG → ∑ acyclic → dataflow → circuits

Critical Paths

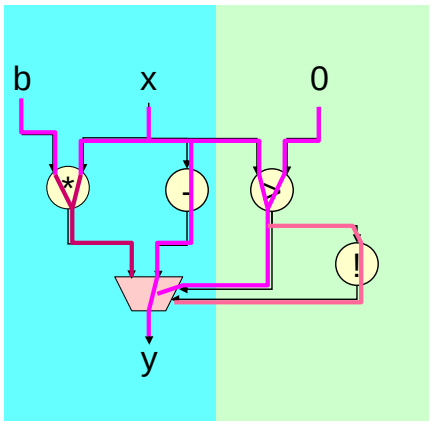
```
if (x > 0)
        y = -x;
else
        y = b*x;
```



C → CFG → ∑ acyclic → dataflow → circuits

Lenient (Early) Evaluation

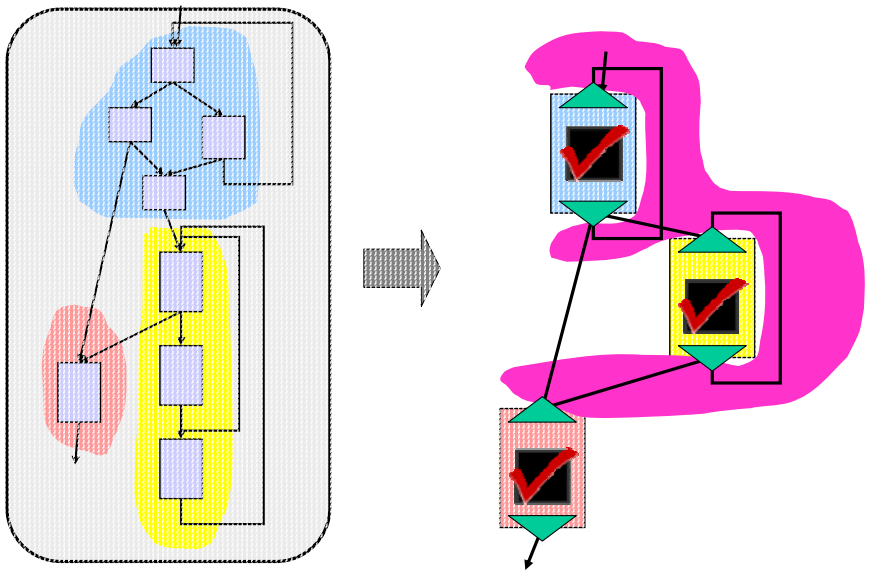
```
if (x > 0)
        y = -x;
else
        y = b*x;
```



Solves the problem of unbalanced paths

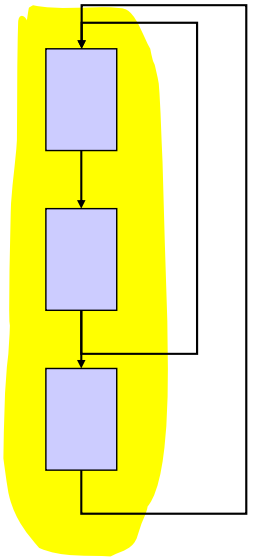
C → CFG → ∑ acyclic → dataflow → circuits

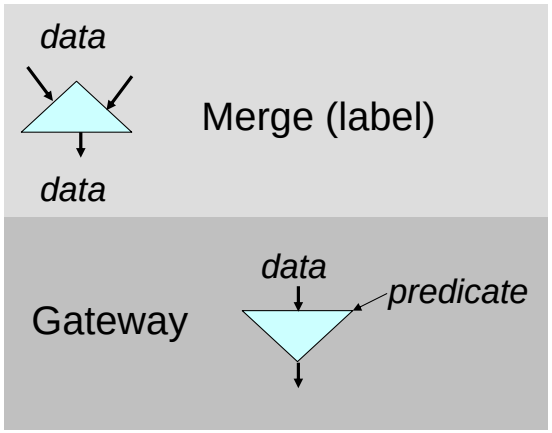
Stitching Hyperblocks



C → CFG → ∑ acyclic → dataflow → circuits

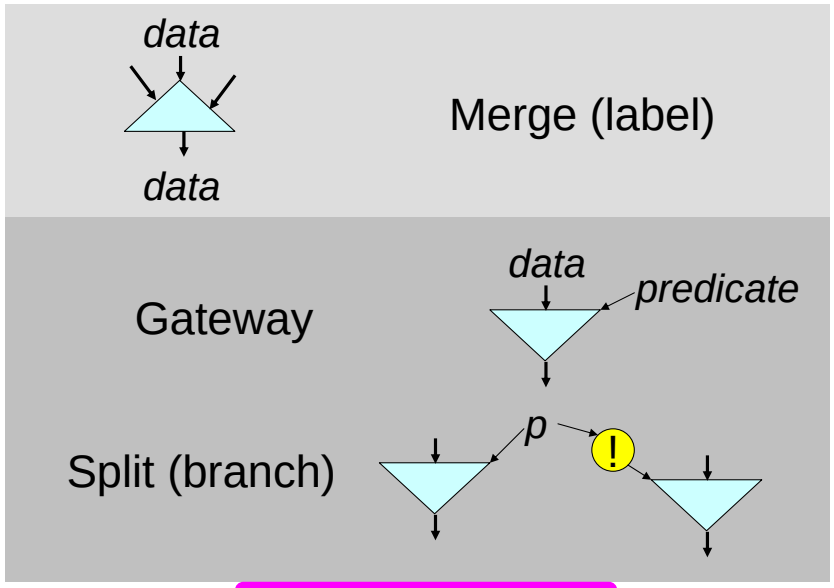
Loops: Control Flow ⇒ Data Flow





C → CFG → ∑ acyclic → dataflow → circuits

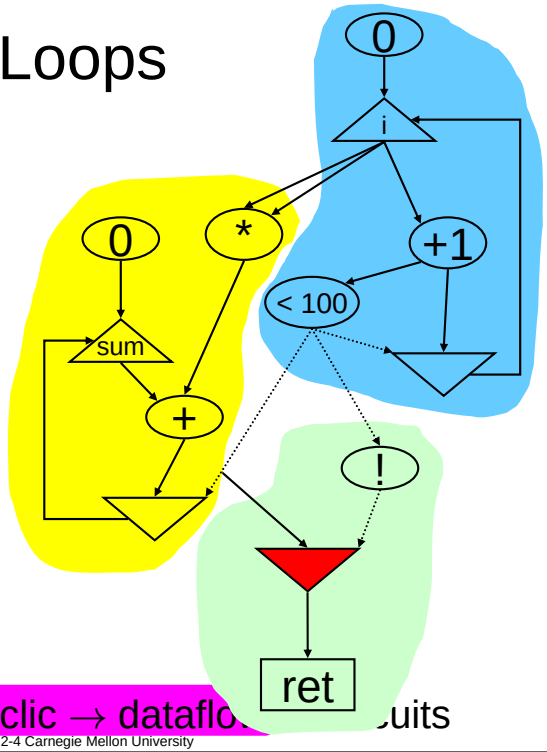
Control Flow ⇒ Data Flow



C → CFG → ∑ acyclic → dataflow → circuits

Loops

```
int sum=0, i;
for (i=0; i < 100; i++)
    sum += i*i;
return sum;
```

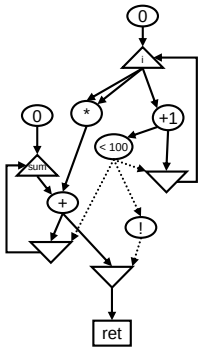


C → CFG → ∑ acyclic → dataflo… …uits

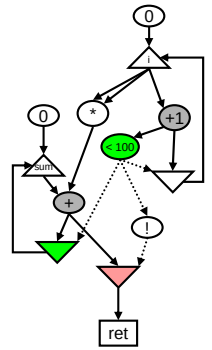
Interlude

Visualization and Debugging

Visualization

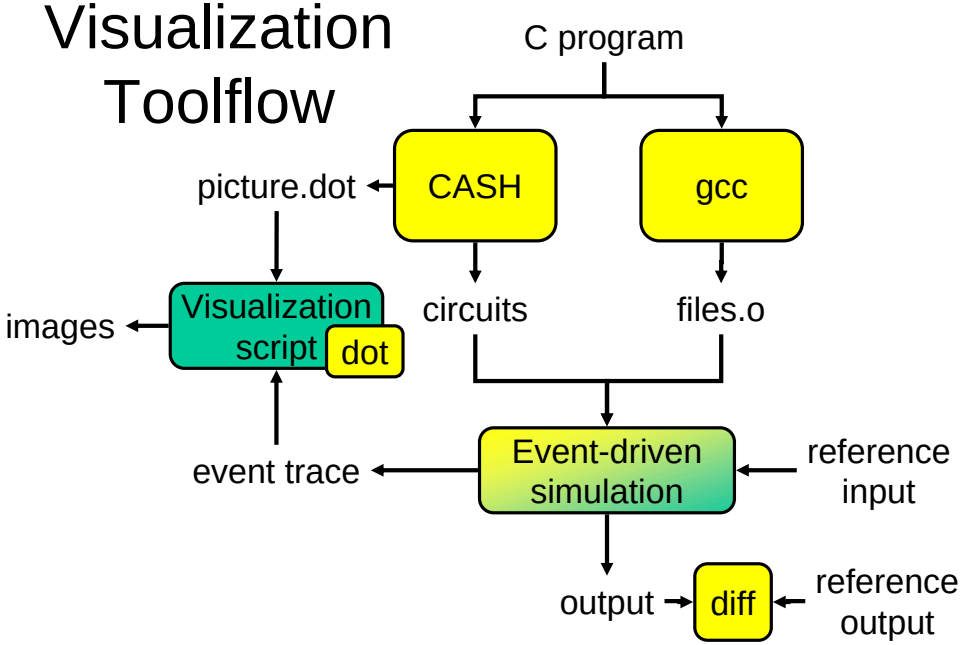



```
0.out[0] produces now 7.in[2]
0.out[0] produces now 14.in[0]
1.out[0] produces now 15.in[0]
2.out[0] produces now 16.in[1]
 Event 0: 7.in_ready[2] posted by 0
 Event 1: 14.in_ready[0] posted by 0
  Op 14 ready to execute
 Event 2: 15.in_ready[0] posted by 1
  Op 15 ready to execute
 Event 3: 16.in_ready[1] posted by 2
  Op 16 ready to execute
Executing 16 {0} (op_mu)
Executing 15 {0} (op_mu)
Executing 14 {0} (op_mu)
squares: op   0/op_arg
squares: op   7/op_eta    {1}
squares: op  14/op_mu     {1}
squares: op  15/op_mu     {1}
squares: op  16/op_mu     {1}
```



Circuit picture | Execution trace | Visualization

Visualization Toolflow



Visualization Options

Visualization script	c2verilog -a *circuit animation*	c2verilog -p *dynamic critical path*
Event-driven simulation	C + runtime lib *high-level*	Verilog+ModelSim *low-level*

Live Demo #1

compiling and running “sum of squares”

Initial Setup

1. Start X-Win (double click on the icon)
2. Start putty
   - Select “troia” connection (double click on the name)
   - Login name: tut*X* (*X* = number on the machine CAD_PC*X*)
   - Password: async04
3. In the putty window, type:

   $ initial.pl cash

   $ cd cash

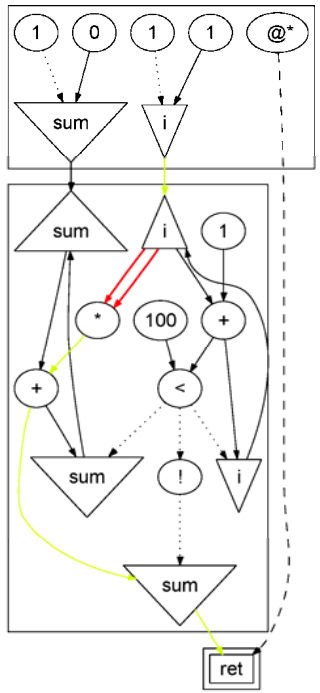
Compiling Sum of Squares

- Look at source: demo_squares/orig/sum_of_squares.c
- Compile and simulate (high-level):

  $ c2verilog.pl –p squares1.dot demo_squares squares
- This generates
  - a dot file (squares1.dot)
  - with critical path highlighted
- View the circuit using dotty

  $ dotty squares1.dot &
  - Use the Dotty hand-out to interpret the graph
  - Note the critical path
    - red = most critical
    - green = less critical

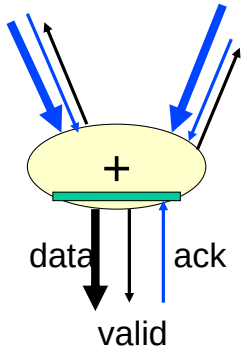
Dotty output for squares



Animation of the circuit

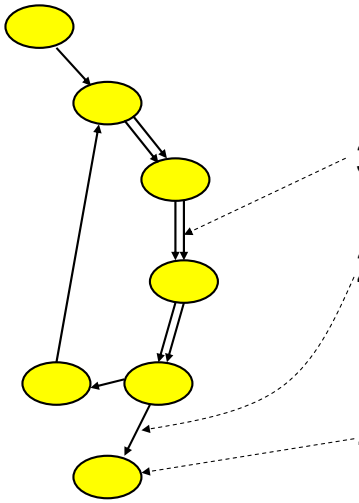
- Break for ghostview

Last-Arrival Events



- Event enabling the generation of a result
- May be an ack
- Critical path=collection of last-arrival edges

Dynamic Critical Path

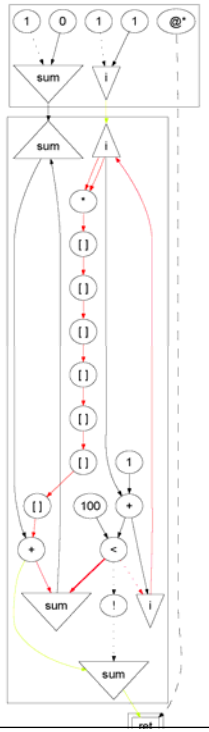


3. Some edges may repeat
2. Trace back along last-arrival edges
1. Start from last node

Pipelining the multipliers

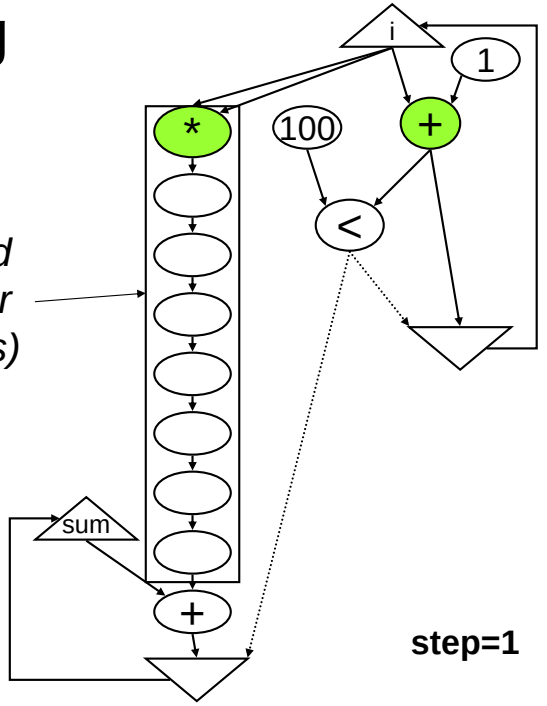
- Compile and simulate (high-level):
  - $ c2verilog.pl **–m** –p squares2.dot demo_squares squares
- View the circuit using dotty
  - $ dotty squares2.dot &
  - – Use the Dotty hand-out to interpret the graph
  - – Note the critical path (colored edges, red = most critical)

After pipelining the Multiplier



Pipelining

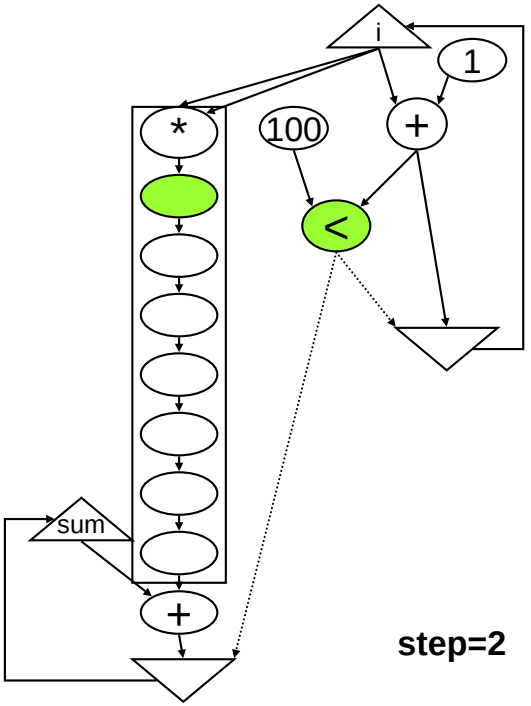
pipelined multiplier (8 stages)




```
int sum=0, i;
for (i=0; i < 100; i++)
    sum += i*i;
return sum;
```

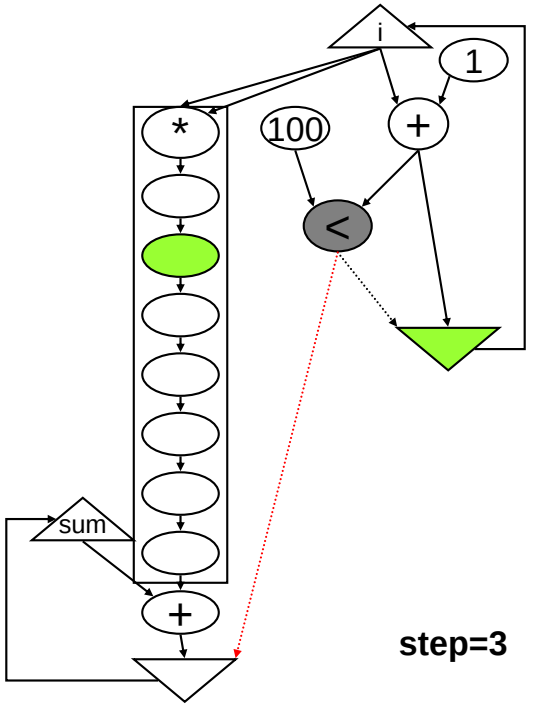
step=1

Pipelining



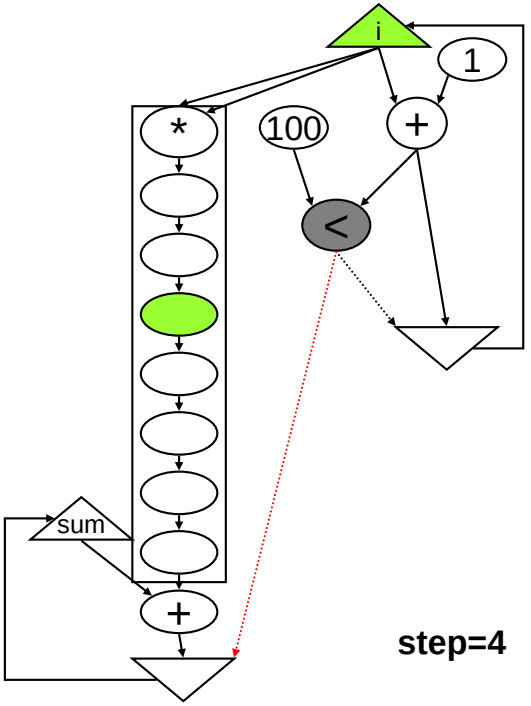
step=2

Pipelining



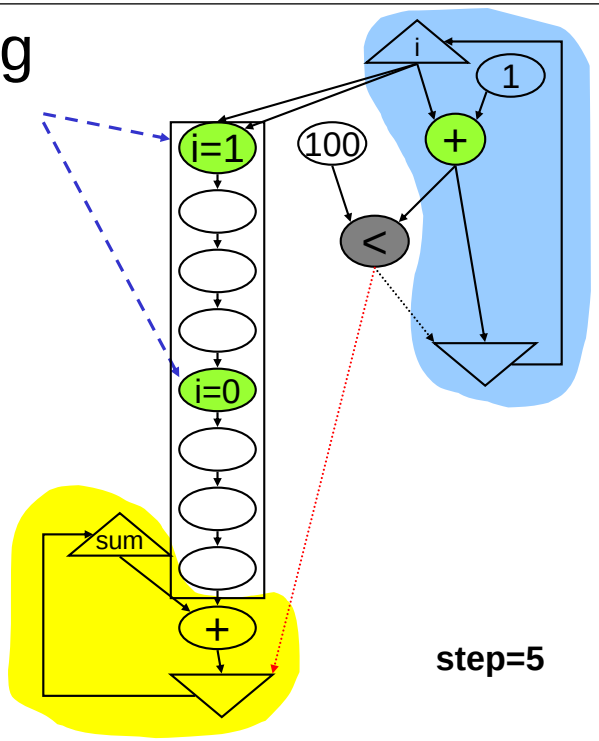
step=3

Pipelining



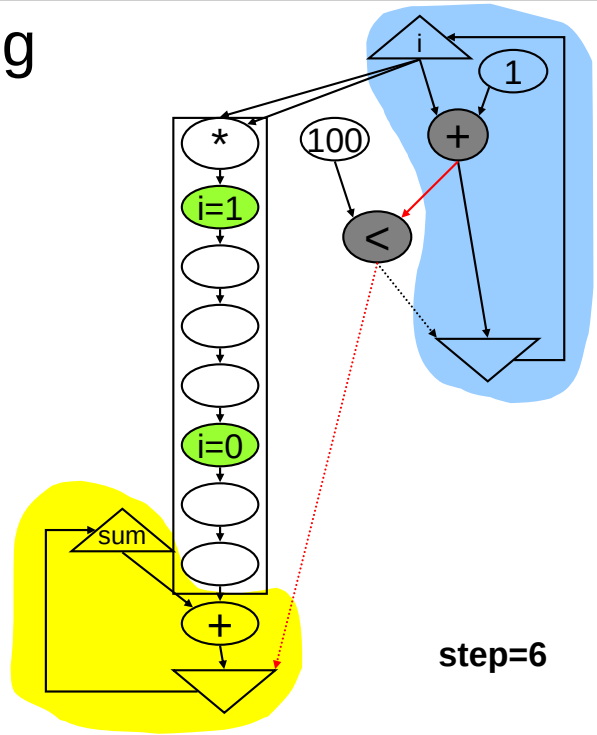
step=4

Pipelining



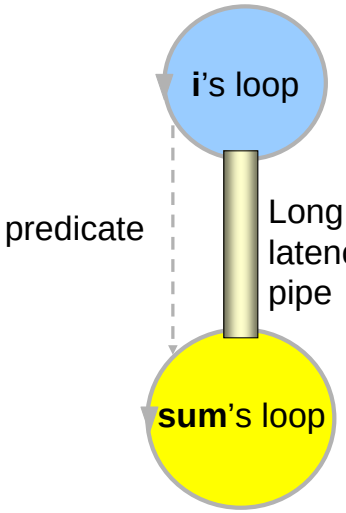
step=5

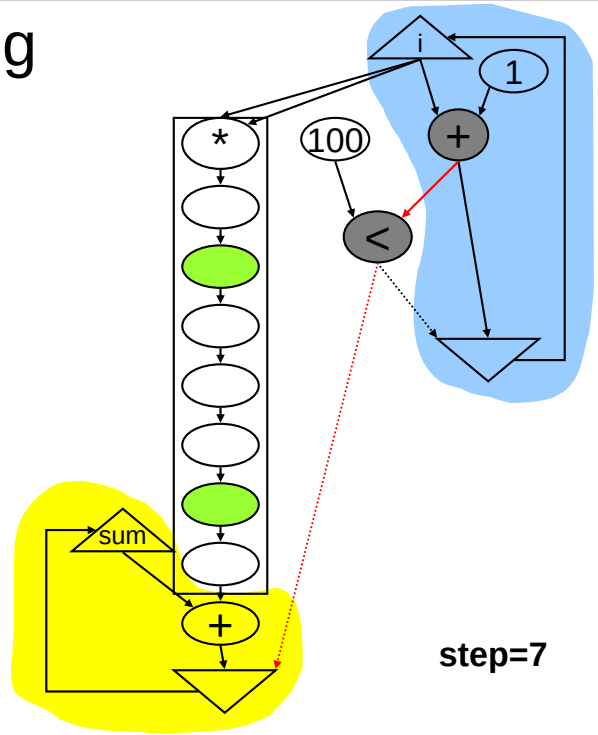
Pipelining



step=6

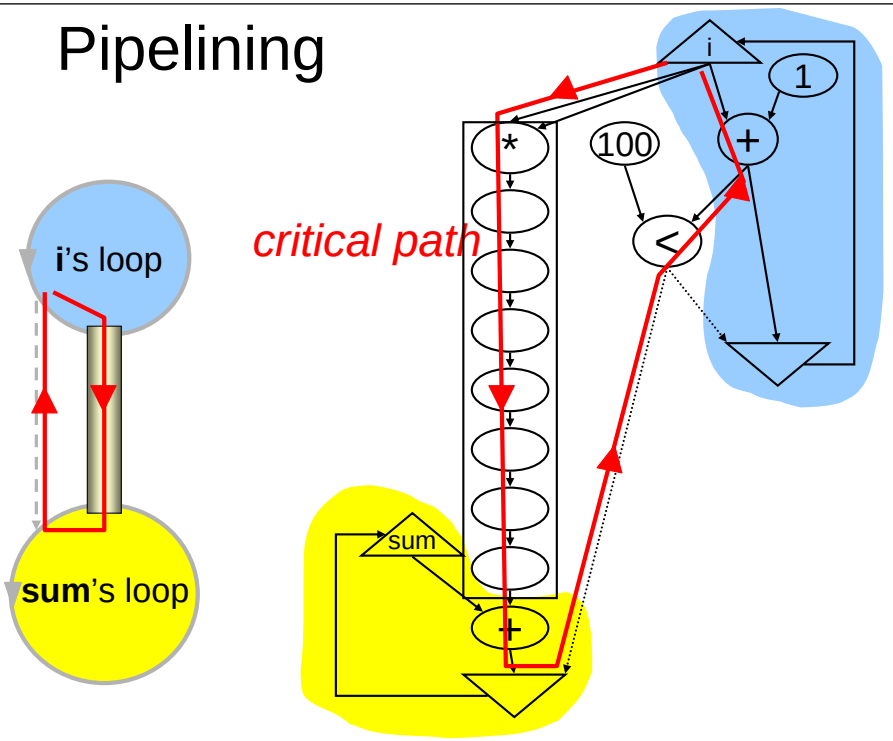
Pipelining



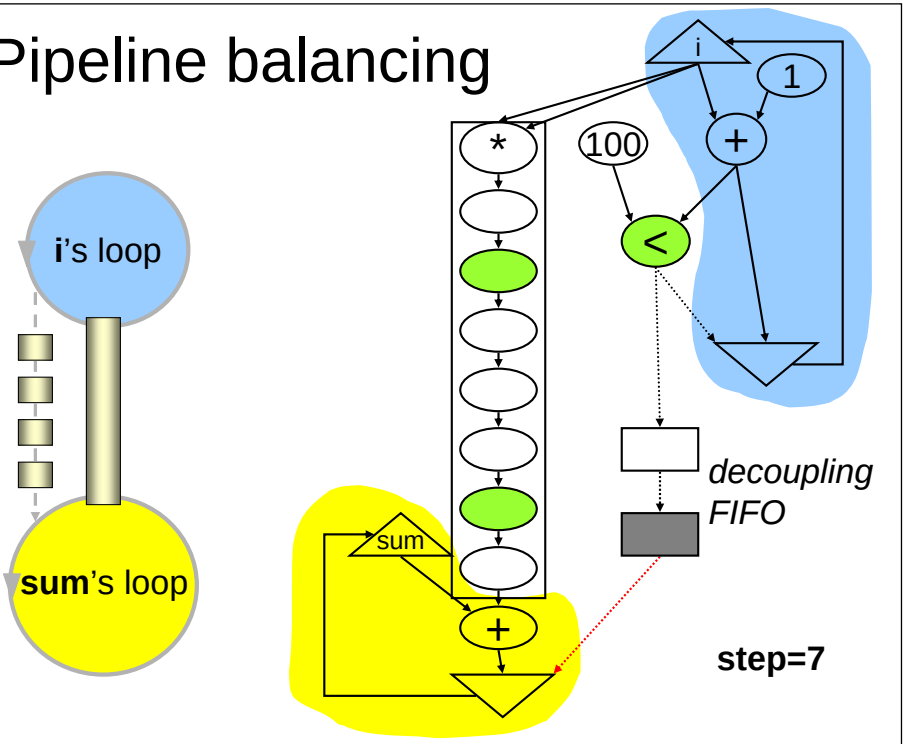


step=7

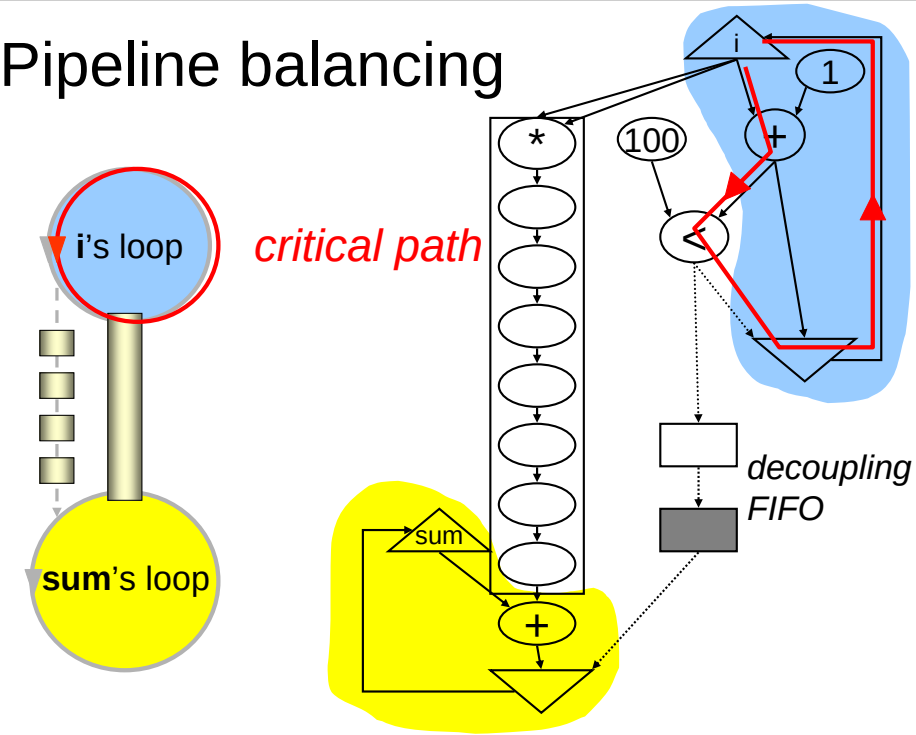
Pipelining



Pipeline balancing



Pipeline balancing



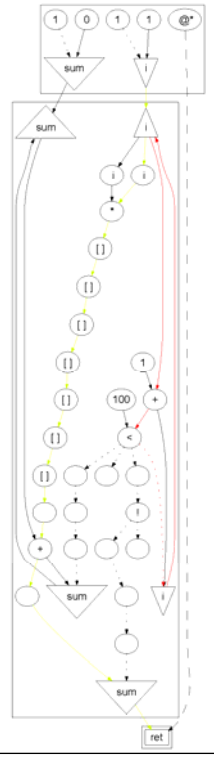
Live Demo #2

Pipeline Balancing "sum of squares"

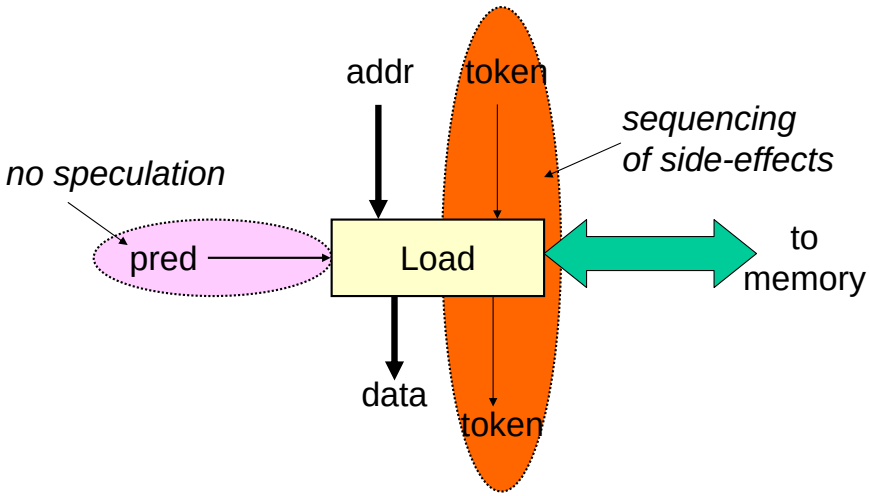
Compile with ‘-i’ option

- Compile and simulate (high-level):
  $ c2verilog.pl –m **–i** –p squares3.dot demo_squares squares
- View the circuit using dotty
  $ dotty squares3.dot &
  – Use the Dotty hand-out to interpret the graph
  – Notice the difference in the critical path between this and squares2.dot
  – Empty ovals = FIFO’s for pipeline balancing

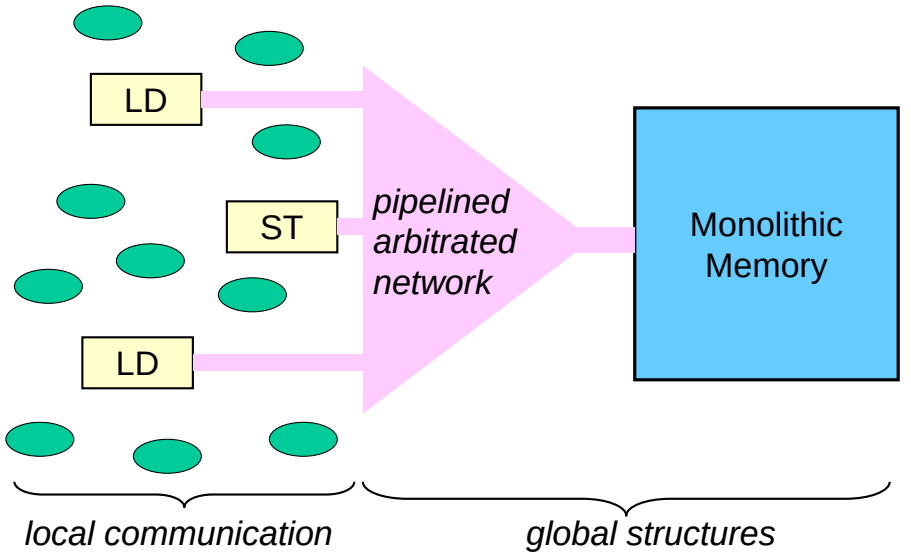
Dotty after balancing



Predication and Side-Effects

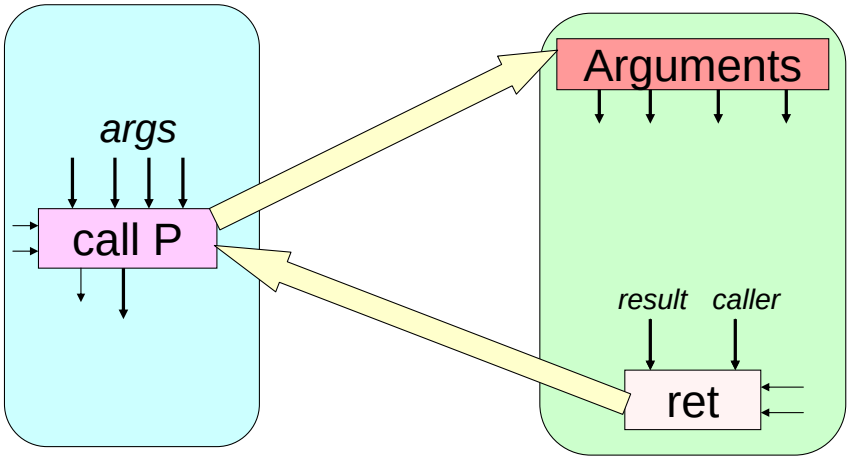


Memory Access



Future work: fragment this!

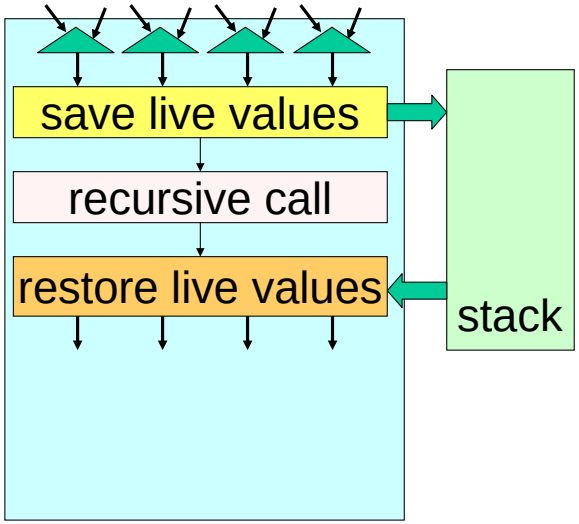
Procedure calls



caller

callee

Recursion



Other details

- Stack handling for locals
- "Return address" for procedures
- Rules for executing "constants"
- Additional synchronization required to handle loops
  (all "merge" nodes must pass same input)
- Hold loop-invariant value flow in registers

$C \rightarrow CFG \rightarrow \sum$ acyclic $\rightarrow$ dataflow $\rightarrow$ circuits

Part II

From Dataflow to Asynchronous Circuits

$C \rightarrow CFG \rightarrow \sum$ acyclic $\rightarrow$ dataflow $\rightarrow$ circuits

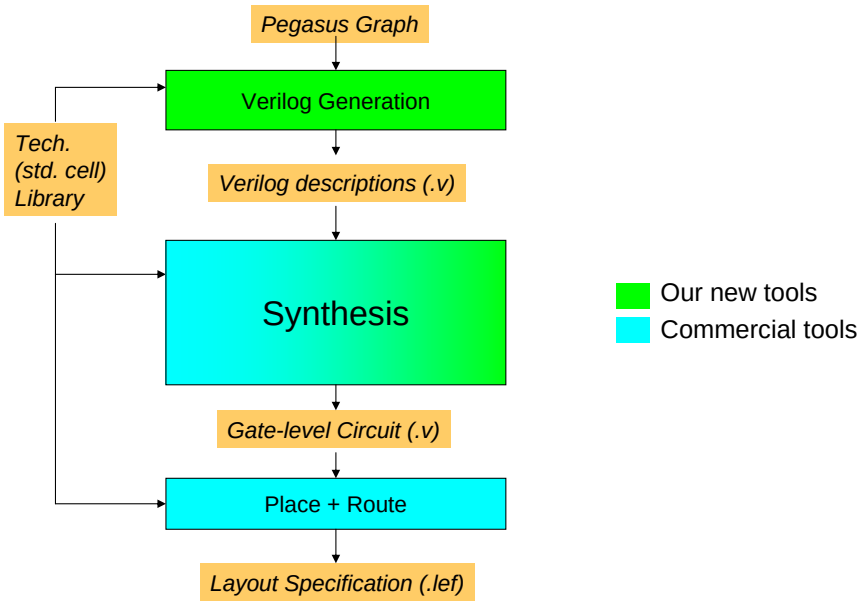
Background Demo

ANSI-C to circuit layout
adpcm_decoder – Voice decoding

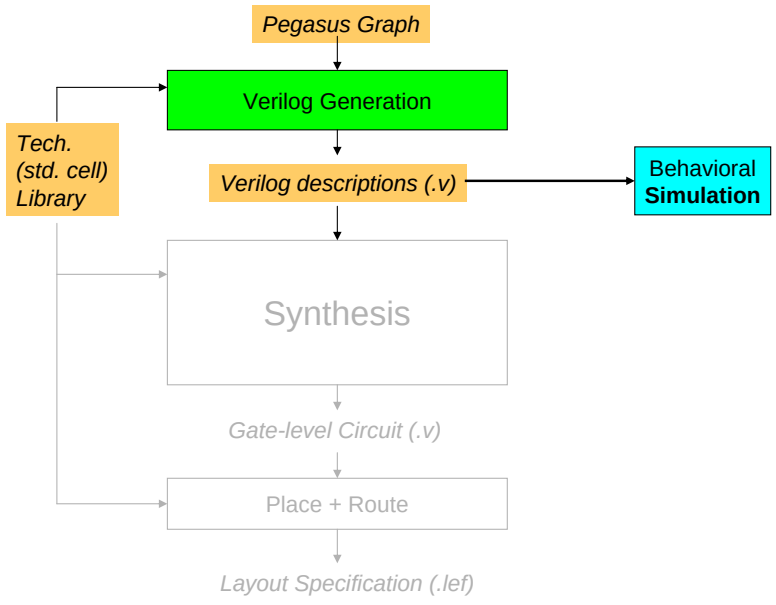
Analyze Datapath Delays

- View the C source code
  - $ emacs adpcm_d/orig/adpcm.c &
- Generate the Verilog
  - $ c2verilog.pl adpcm_d adpcm_decoder
- Move to ece.cmu.edu
  - $ scp –r adpcm_d/verilog/* girishv@entwhistle.ece.cmu.edu:/scratch/girishv/livedemo/vdump
  - $ ssh girishv@entwhistle.ece.cmu.edu
  - $ cd /scratch/girishv/livedemo
- Compute datapath delays
  - $ generate_delay_file.pl –k –d vfiles vdump delays

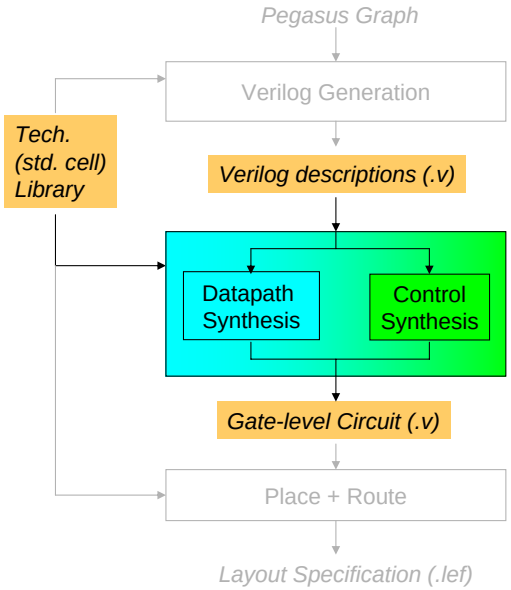
Overview: Design Flow



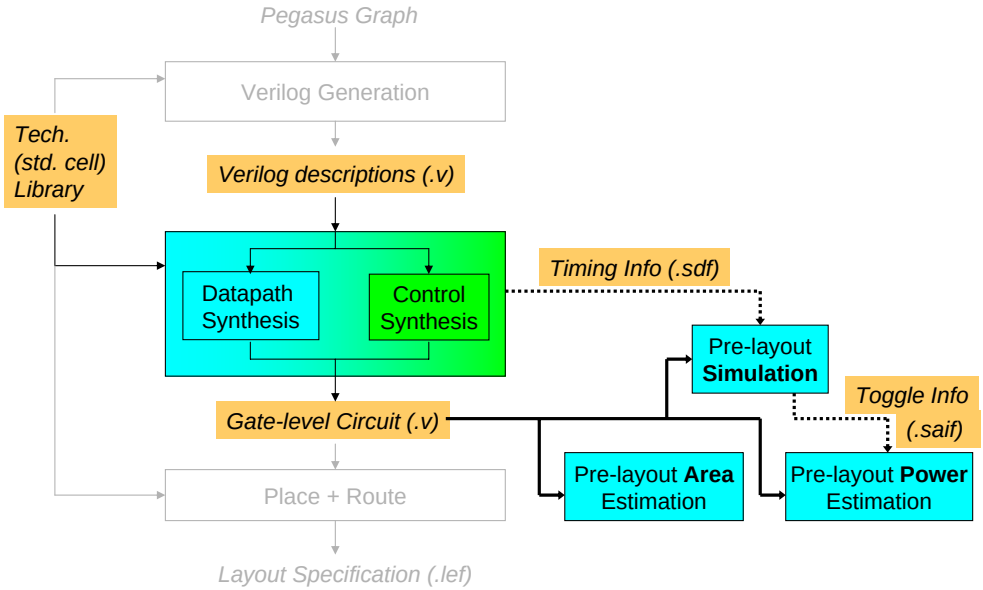
Design Flow: Behavioral Synthesis



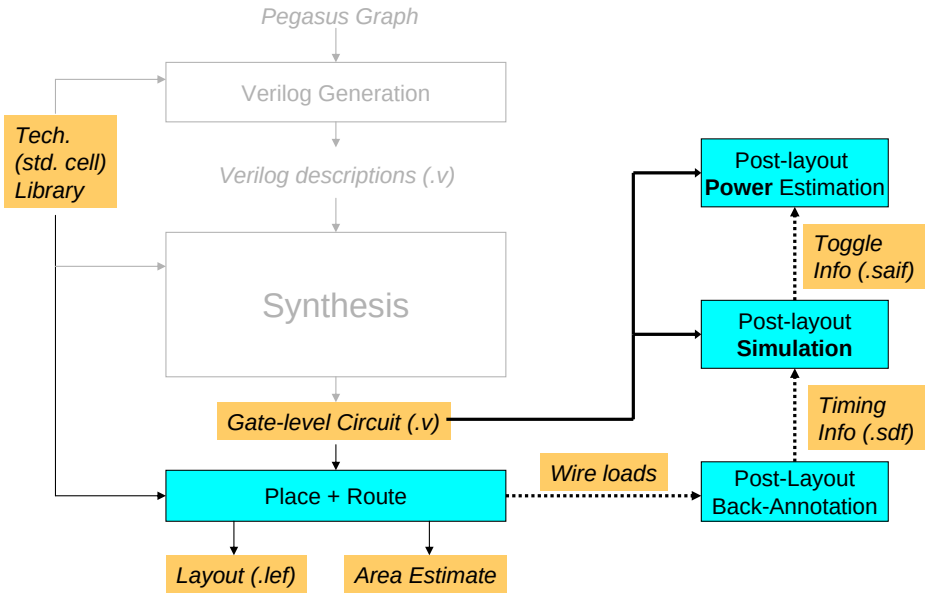
Design Flow: Synthesis



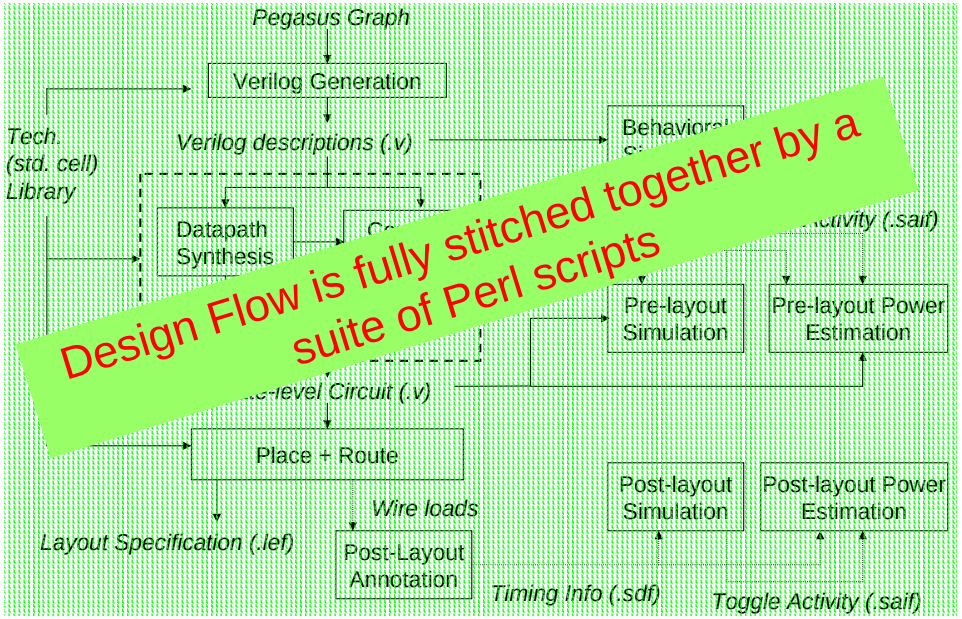
Design Flow: Synthesis



Design Flow: Layout



Design Flow: Integration & Automation



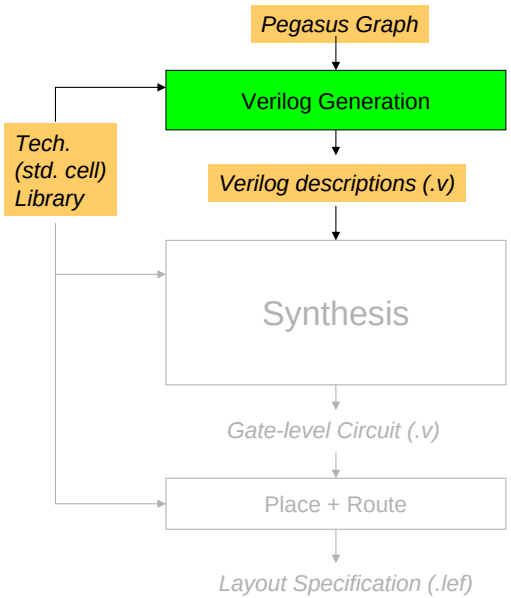
Background Demo: Check

ANSI-C to circuit layout
adpcm_decoder – Voice decoding

Synthesis

- View the delay info
  - $ cat delays
  - $ cat vfiles/op_add.v
- Insert matched delays
  - $ insert_delays.pl delays vfiles delaydir
- Tech-Mapping using Synopsys Design Compiler
  - $ techmap_bench.pl –m all –x delaydir syndir

Outline: Verilog Generation



Pegasus ⇒ Verilog

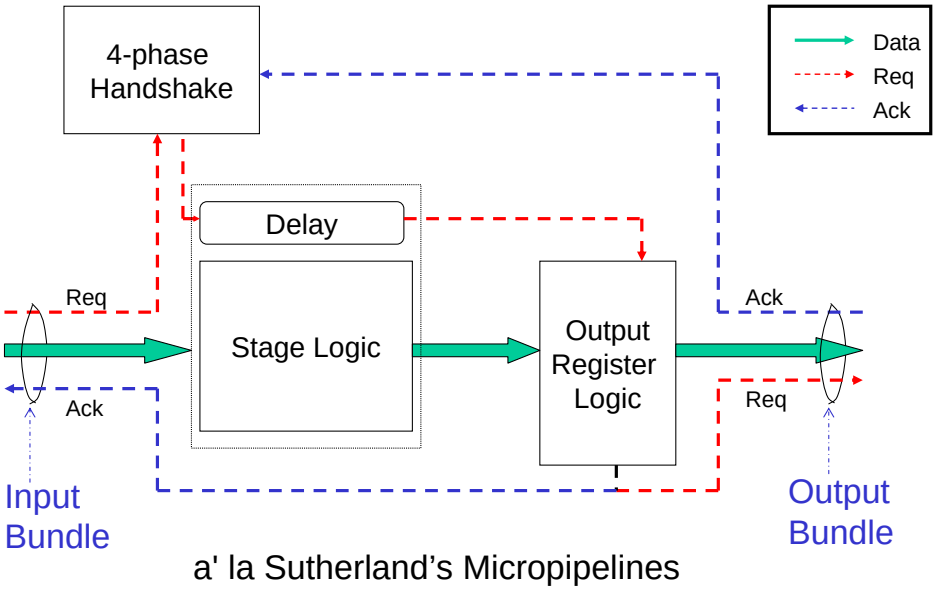
Implementation: 1:1 mapping

- Node ⇒ pipeline stage
- Edge ⇒ bundled-data channel
  - Communication: *4-phase bundled data protocol* allows for reusing existing synchronous datapath units

Other Features:

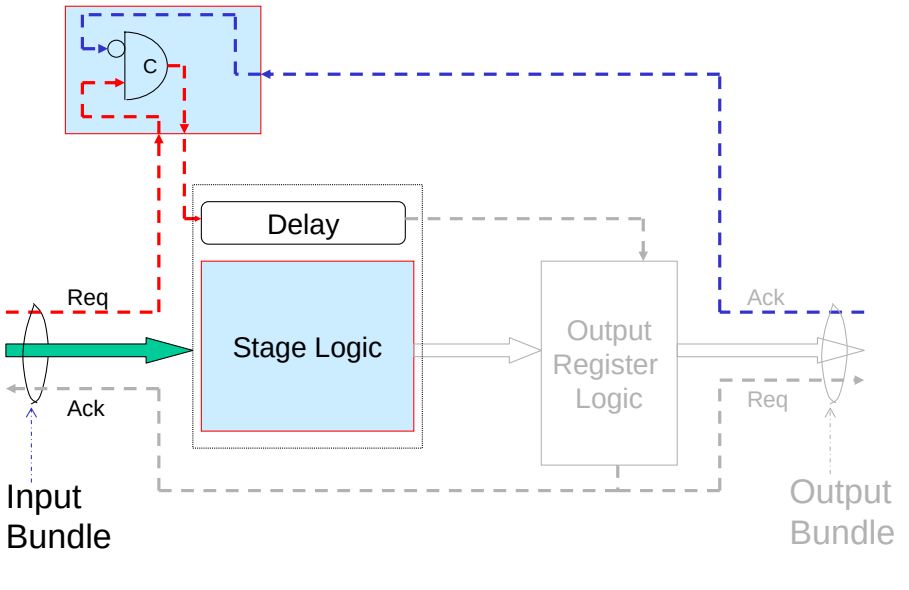
- Some special implementations (eg. leniency)
- Memory: shared
  - needs arbitration network

Pipe Stage Architecture

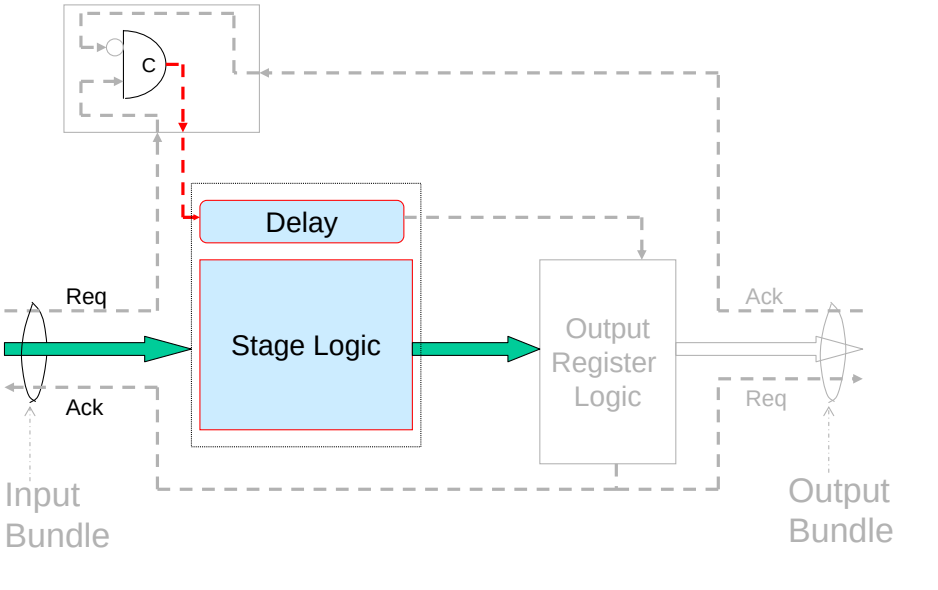


a' la Sutherland's Micropipelines

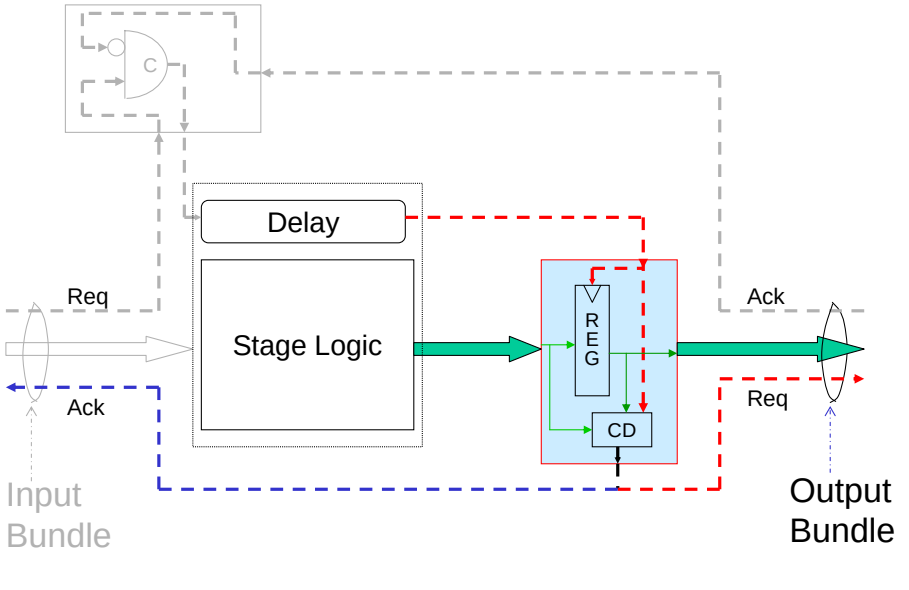
Pipe Stage Architecture



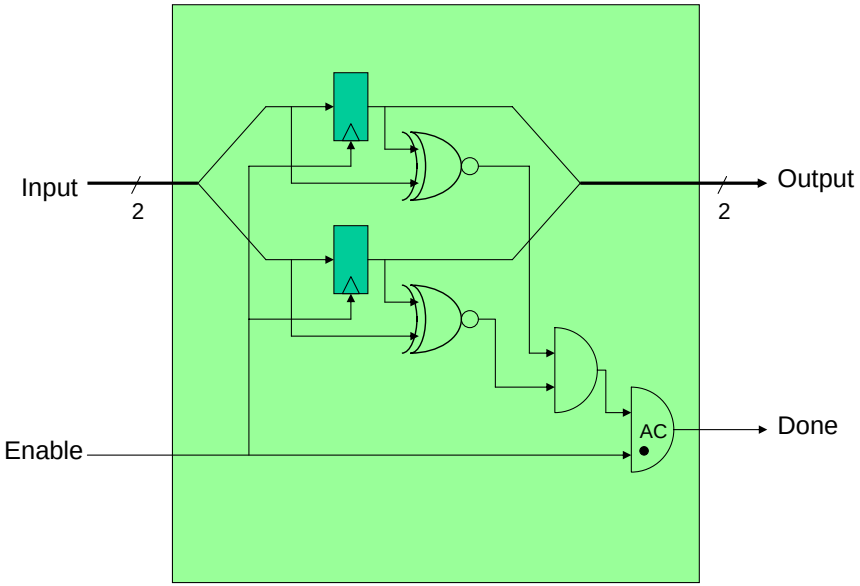
Pipe Stage Architecture



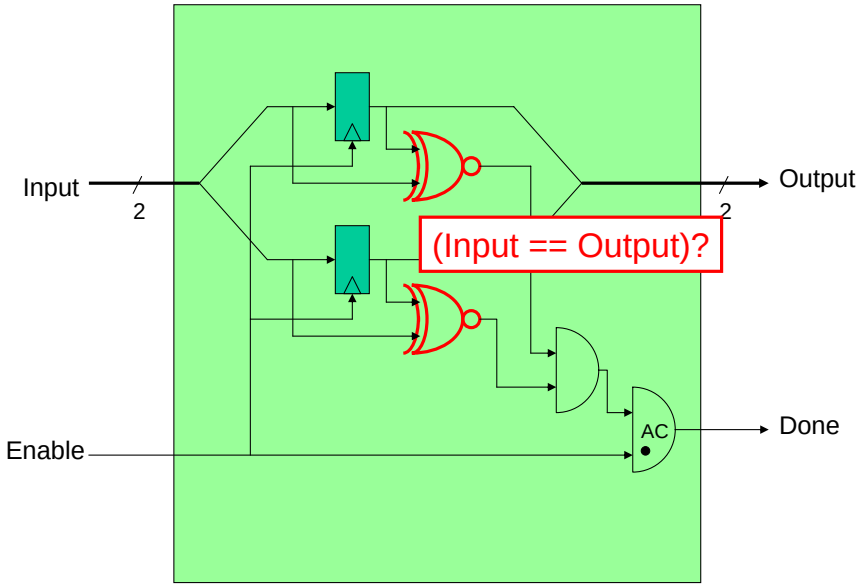
Pipe Stage Architecture



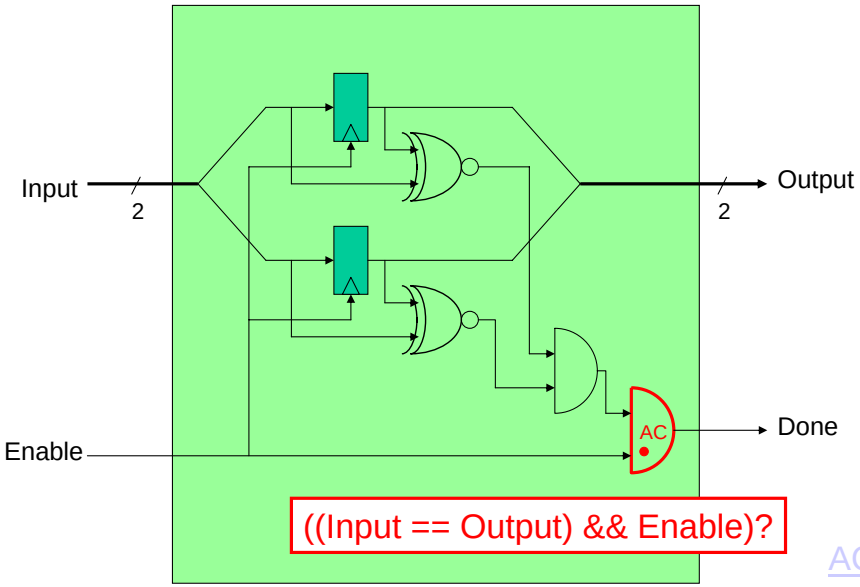
Completion Detection: 2-bit Register



Completion Detection: 2-bit Register

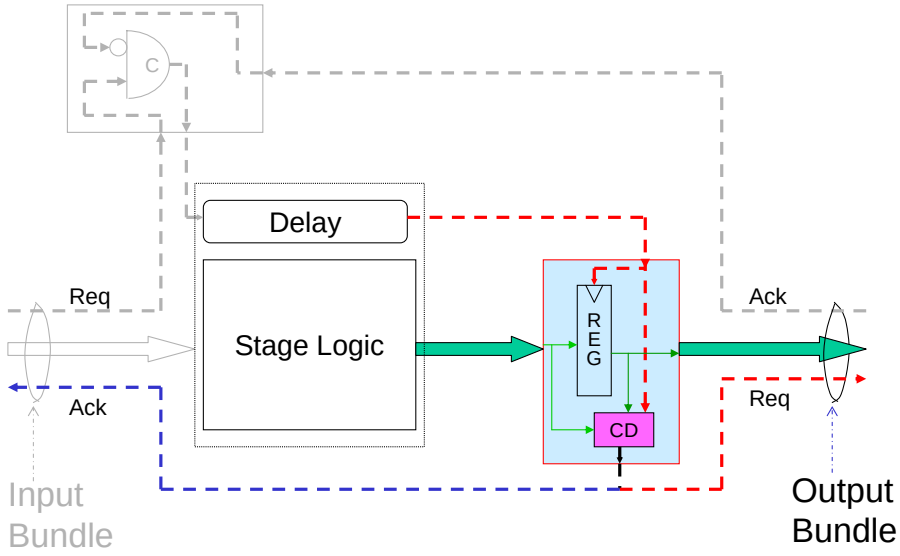


Completion Detection: 2-bit Register



AC

Pipe Stage Architecture



Background Demo: Check

ANSI-C to circuit layout
adpcm_decoder – Voice decoding

Simulation & Layout

- See synthesis generated files
  - $ ls syndir/*.sdf syndir/*.v
  - circ_gate.v is the gate-level verilog
- Simulate
  - $ vsim_bench_tut.pl –x delaydir syndir 100000 simdir
- Note (and save) timing
  - $ grep “donetoken = 1” transcript
  - $ cp transcript pre_layout_sim.time
- Do Layout
  - $ layout.pl –l wireload.cse –m all syndir pnrdir
  - Follow instructions (“seultra –m=3500”)

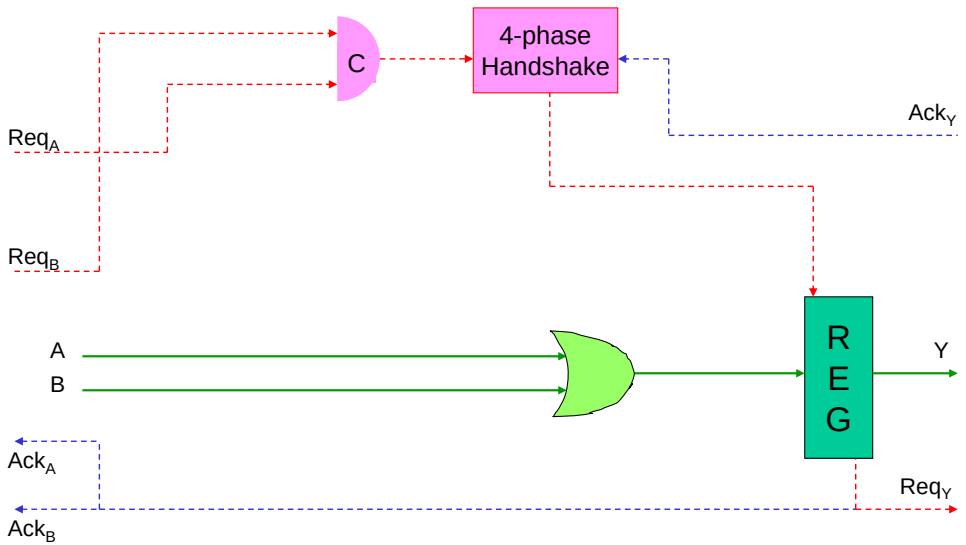
Lenient (Early) Evaluation

Leniency: output result before all inputs arrive

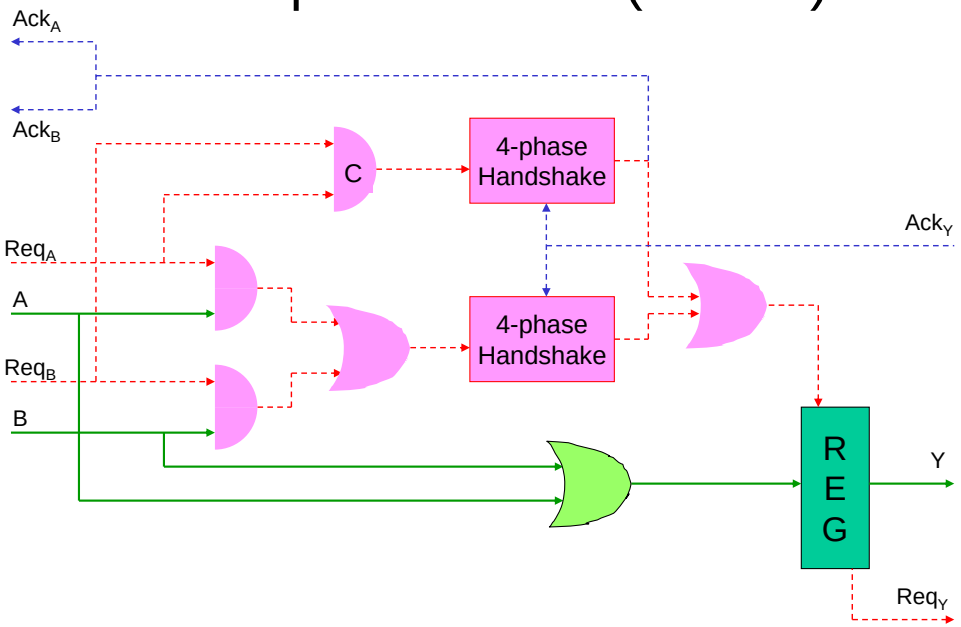
Handshaking:

- Output *Req* is emitted early
- Input *Acks* emitted after all inputs arrive

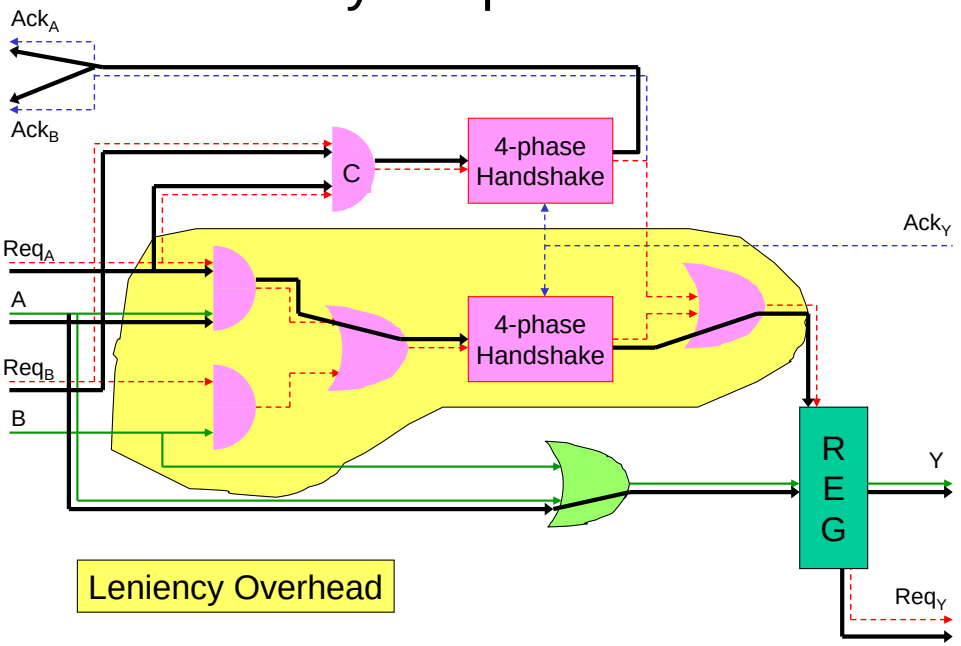
Example: 1-bit OR (strict)



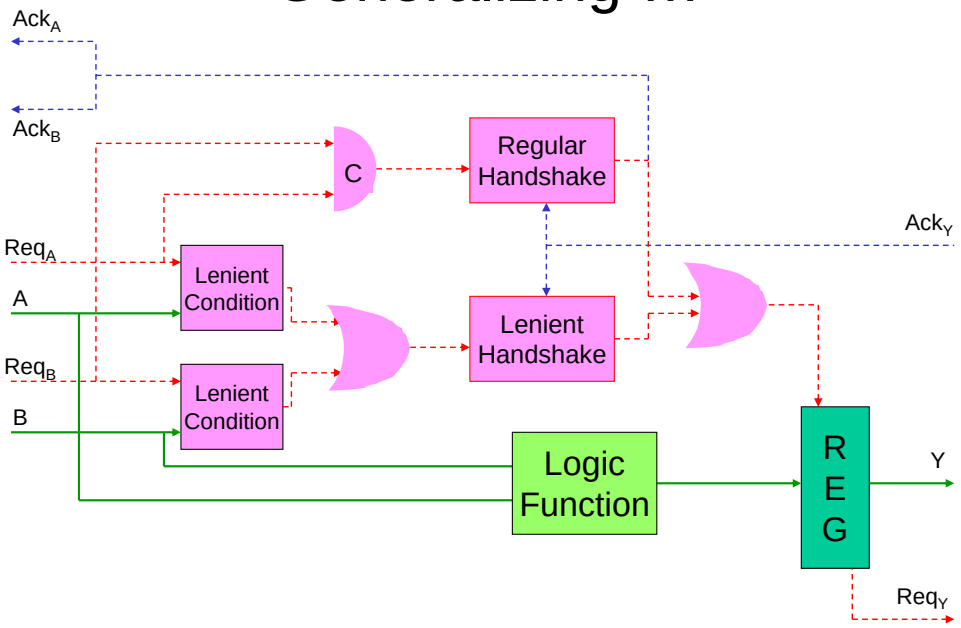
Example: 1-bit OR (lenient)



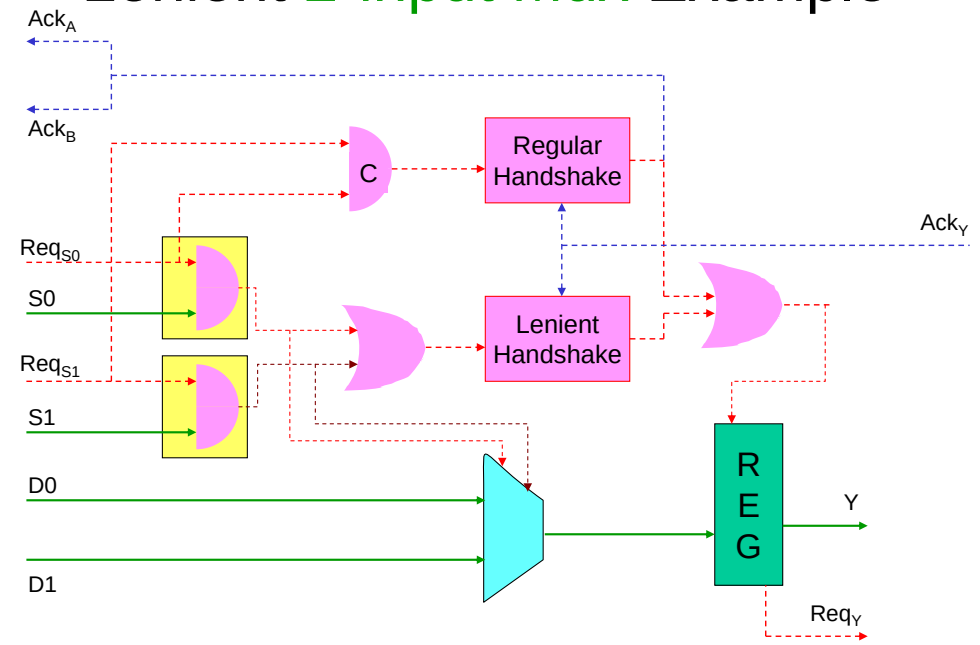
Early Output: A = 1



Generalizing …



Lenient 2-input Mux Example



Live Demo #3

Leniency in adpcm_d

Run without leniency

- Compile
  - $ c2verilog.pl –v adpcm_d adpcm_decoder
- Simulate using Verilog-XL
  - $ cd adpcm_d/sim_ver/
  - $ verilog +gui *.v &
  - $ Exit the simulator (close windows)
  - $ cd ../../
- Note and save the simulation time
  - $ timetaken.pl adpcm_d

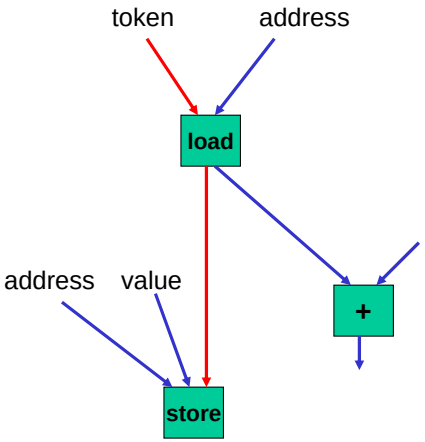
Run with leniency

- Compile
  - $ c2verilog.pl **–l** –v adpcm_d adpcm_decoder
- Simulate using Verilog-XL
  - $ cd adpcm_d/sim_ver/
  - $ verilog +gui *.v &
  - $ Exit the simulator (close windows)
  - $ cd ../../
- Compare the simulation time with strict execution
  - $ timetaken.pl adpcm_d

Memory Accesses

- **Implementation**: shared, monolithic memory
  - Many access points, one destination
  - CAB synthesizes an arbitration network for access to memory
- **Token forwarding**: ensures program order for memory operations
  - Source memory op must be *issued before* destination memory op for tokens

Example

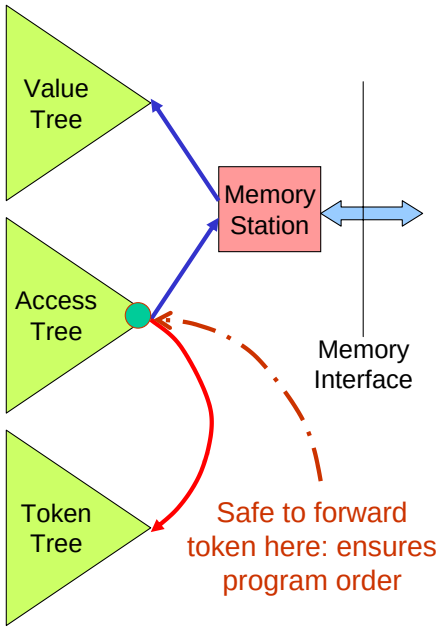


- Sample graph
- Intervening memory access between `load` and + nodes
- `store` must be issued after `load`

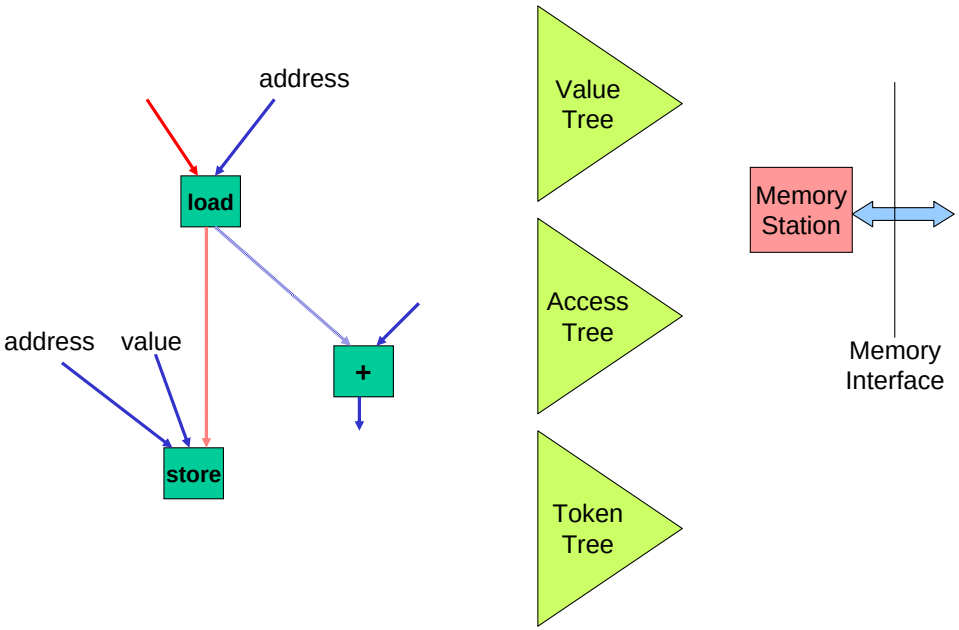
Memory Access Structures

Four Structures (pipelined):

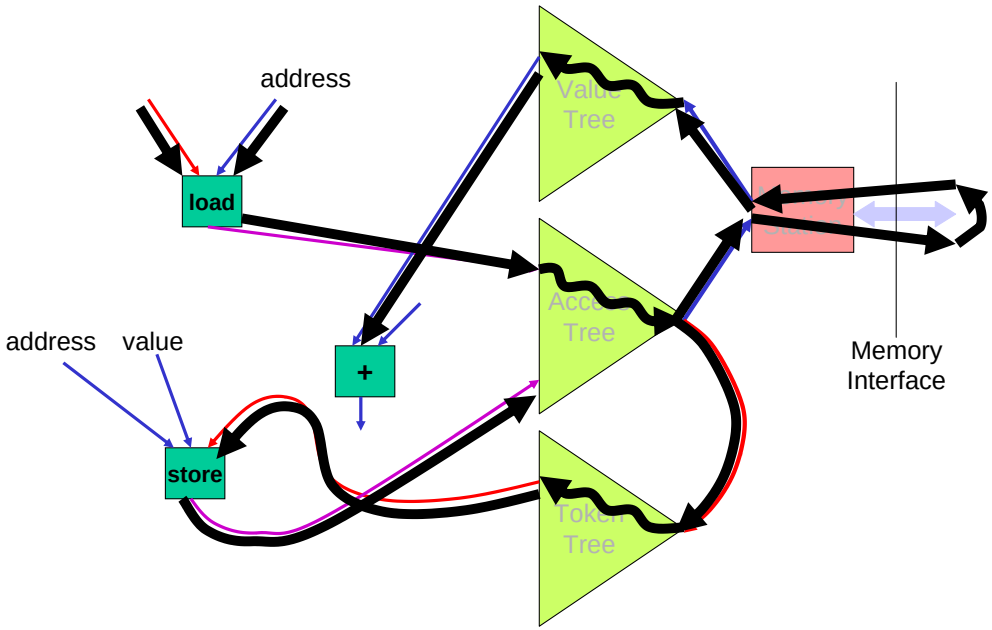
- *Memory Station*: interface with memory
- *Access Tree*: arbitrated network for memory access
- *Token Tree*: return network for tokens
- *Value Tree*: return network for values



Back to Example



Back to Example

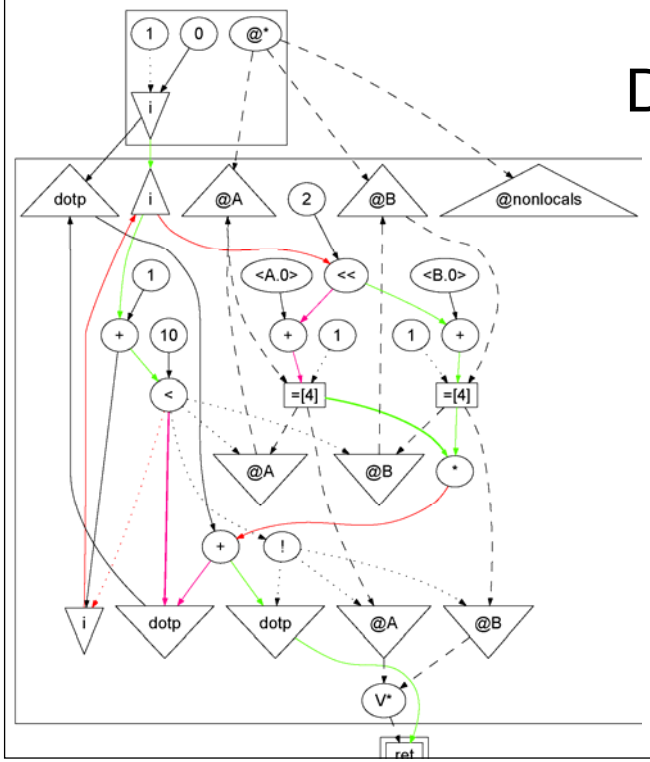


Live Demo #4

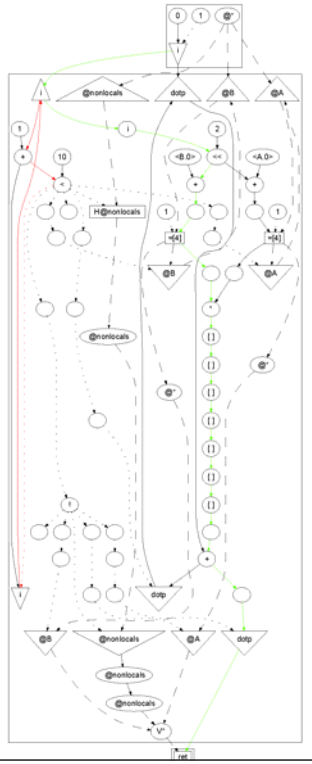
compiling and running “dot product”

Compiling Dot product

- View C source
  - $ cat demo_dot_product/orig/dotp.c
- Compile and simulate (high-level):
  - $ c2verilog.pl –p dotp.dot demo_dot_product dot_product
- View the circuit using dotty
  - $ dotty dotp.dot &
  - Use the Dotty hand-out to interpret the graph
- If you want to generate verilog (not required):
  - $ c2verilog.pl –v demo_dot_product dot_product
  - demo_dot_product/sim_ver/circ_df.v – main circuit
  - demo_dot_product/sim_ver/circ_mem.v – memory arbitration structures
- Simulate using Verilog-XL
  - $ cd dot_product/sim_ver
  - $ verilog +gui *.v
  - Exit the simulator (close windows)

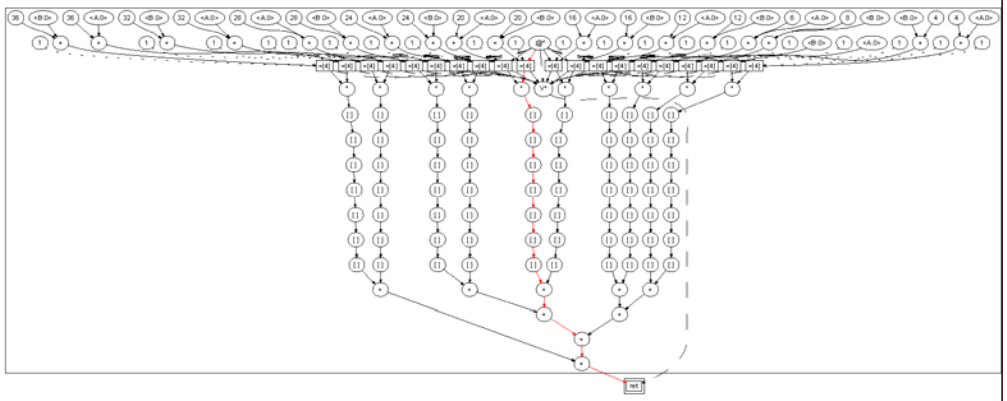


Dotproduct no opts



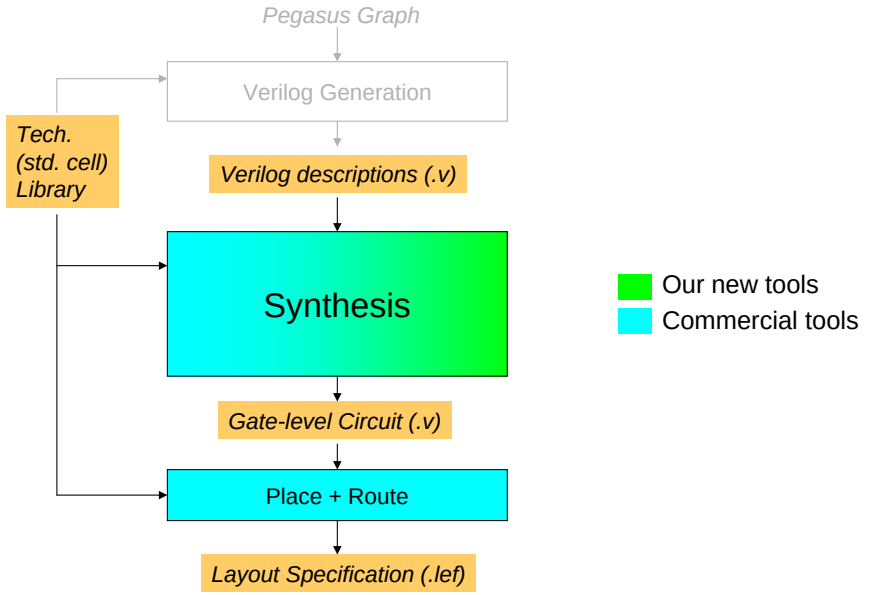
Dotproduct w/balancing

Dotproduct w/unrolling



(The compiler will completely unroll if the body of the loop is small enough and the bounds are known, otherwise partial unrolling is performed.)

Outline: Synthesis and Layout



Background Demo: Check

ANSI-C to circuit layout

adpcm_decoder – Voice decoding

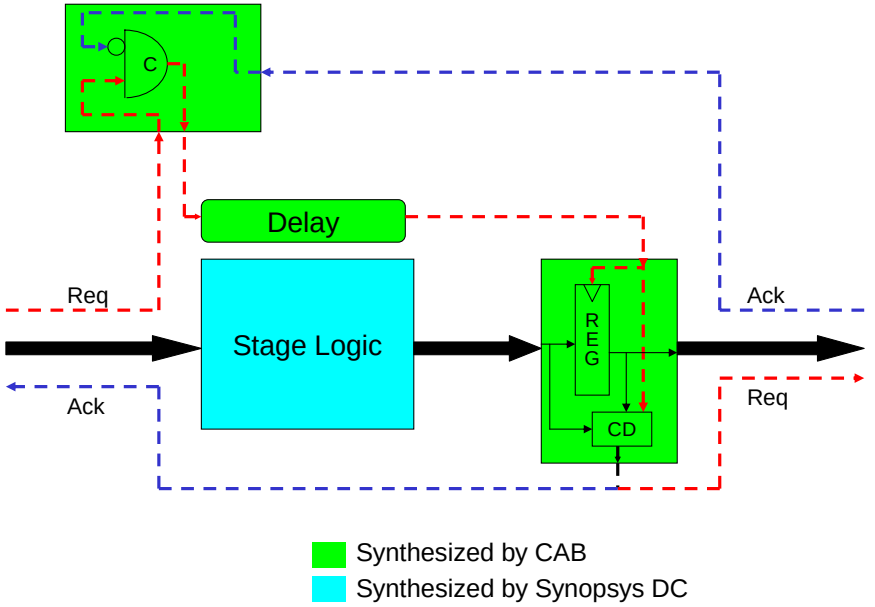
Post-Layout

- Check wire load file
  - $ ls –l wireload.cse
  - $ head wireload.cse
- Back-annotate wire-loads, and simulate
  - $ finish_layout_dir.pl –m all –x simdir wireload.cse layoutsim

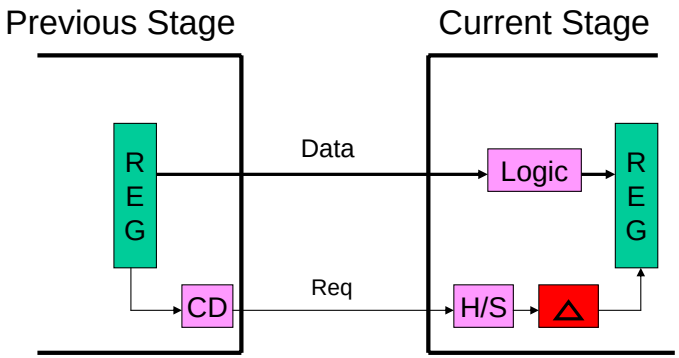
Synthesis and Tech Mapping

- **Hierarchical Synthesis**
  - Each pipeline stage is individually synthesized
  - Circuit is then structurally composed
- **CAB synthesizes**:
  - Control path Logic (handshaking)
  - Delay Elements
  - Output Registers and Completion Detection logic
- **Synopsys Design Compiler used**:
  - To synthesize datapath logic (in each stage)
  - Delay characterization (at the gate level)
  - Power Estimation (at the system level)

Synthesis of a Pipeline Stage



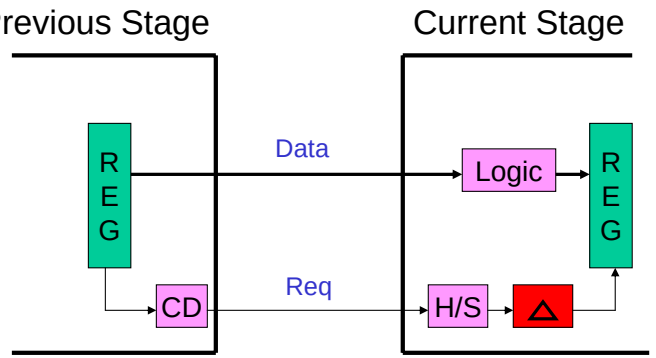
Delay Matching



$\triangle = D_{logic} - (D_{CD} + D_{HS}) + slack$

D_{logic} can be accurately estimated since:
- Logic circuit is localized
- Logic output has a fanout of exactly ONE
- Wire load on the output port is known

Bundling Constraint



Is Data valid when Req is asserted?

- Yes; (at the previous stage's output due to CD)
- Between stages, the onus is on the layout tool

Completion Detection

Place and Route

- No clock tree generation
- Predominantly short, local wires
- Timing dependencies are also local
- Our experience with Mediabench:
  - No timing violations
  - Sometimes synthesis (pessimistically) over-estimates wire loads

Sync Tools for Async Design

Sync tools in our design flow:

- Synopsys' Design Compiler:
  - Datapath synthesis
  - Annotation for delays (pre- and post-layout)
  - Power characterization
  - Area (pre-layout)
- Cadence's Silicon Ensemble:
  - Layout
  - Wire loads and parasitics extraction
  - Area

Sync Tools for Async Design

Conclusions:

- Using bundle-data communication:
  - Re-use synchronous datapath elements
  - Allows for efficient delay matching insertion
- No control logic synthesized!
  - Reason: hazards may be introduced
  - Control logic is very simple; hand-implemented
- Layout:
  - No bundling constraint broken
  - Unexpected positive experience… so far!

Sync tools can be used for async design!

Background Demo: Finally ..

ANSI-C to circuit layout

adpcm_decoder – Voice decoding

Compare Execution Times

- Find Execution times
  $ grep “donetoken = 1” transcript pre_layout_sim.time

Final Demo

Use various compiler options

Compiler Optimizations

- Check compiler options
  $ c2verilog.pl –h
  - -l : Leniency
  - -u : Do unrolling
  - -i : Insert FIFO buffers
  - -O : All of the above
  - -v: Generate verilog output
  - -p: Check critical path
  - -m: Pipeline multipliers
- Use various combinations of these to see the effect on execution time

Instructions for using CASH

- Assume the C file folder is <bench>, and function is <func>
- Compile:
  $ c2verilog.pl [options] <bench> <func>
- If you specify –p <file> above, then you can view the critical path:
  $ dotty <file> &
  – Use the Dotty hand-out to interpret the graph
- If you specify –v <file> above, then you generate verilog
- Verilog Simulation
  $ cd <bench>/sim_ver
  $ verilog +gui *.v

C Program to compile

- Synthesizable function:
  - Must be a leaf function
    - No function calls – incl. printf/scanf/malloc sys calls
  - No Floating Point/Double types/operations
- C file:
  - Include a "main" function to call the synthesizable function
- Directory structure:
  - Create directories $PWD/mycode/orig
  - Put C code in mycode/orig
  - See async_tutorial/doc/* for more information

Conclusions - 1

ASH strengths

Feature	Advantages
No interpretation	Energy efficiency, speed
Spatial layout	Short wires, no contention
Asynchronous	Low power, scalable, tolerant to variation
Distributed	No global signals
Automatic compilation	Design productivity

Conclusions - 2

- Reconfigurable Computing is inevitable
- X-point (Molecular?) switches are ideal for reconfigurable device
- EN imposes new constraints
  - Non-ideal components
  - Regular, homogenous architectures
- EN offers tremendous advantages
  - Billions of devices per cm^2
  - Ultra-low power

Conclusions - 3

- New Abstractions are required
- Abstraction Requirements:
  - Tool-friendly not human-friendly
  - Support parallel research activities
  - Promote interdisciplinary research
- If you pick the right abstraction: Custom Hardware from C is possible!
  - Dusty deck C
  - Automatic translation
  - High performance
  - Low-power circuits

Finally

- We welcome criticism and comments
- We also welcome collaboration
- So, please speak to one or more of us over the next few days
- www.cs.cmu.edu/~phoenix

Verilog File Descriptions

In mycode/sim_ver/

circ_io_wrapper.v	Top-level testbench
mem_wrapper.v	Memory Emulator
circ_df.v	Structural Verilog circuit
circ_mem.v	Memory Access Tree
pegasuslib_gen.v pegasusmem_gen.v	Pegasus Macro libraries
gates_bhv.v macros_bhv.v	Standard Cell library abstraction

Understanding the Dotty Graph

Dotty Key Bindings

‘z’	Zoom out
‘Z’	Zoom in
‘ ‘	Redraw graph
‘L’	Reload graph
‘f’	Find node
‘u’	Undo

Some Common Themes

- Solid arrows represent data wires
- Dashed arrows are token wires
- Dotted arrows are predicate wires
- Bounding box containing a set of nodes represents a hyperblock
- Colored Edges represent the critical path – red being the most critical (and thicker the redness, the more critical)

Constant Wire carrying value *k*

A function argument

A merge operation.
If *Var* begins with a @, then it is a pointer.
@nonlocals refers to the location set of aliases that may interfere in the absence of alias information.

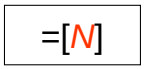
A pipeline stage for a pipelined arithmetic operation.

A gateway operation. (Sometimes drawn as a triangle pointing down.)

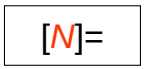
A token merge operation. Collects all input tokens, before releasing output token.

A hold operation. It represents loop-invariant data that is held in this op until the loop completes. It corresponds to variable *Var* in the source program

Static Address of variable *Var* in the program. This refers to arrays that have been statically allocated in the source C program, and whose addresses are known at compile-time. *N* represents the offset of array *Var*

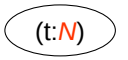


Load. Loads an *N*-byte value from memory



Store. Stores an *N*-byte value into memory

Multiplexor. The output of mux is the variable *Var* in the source C program.

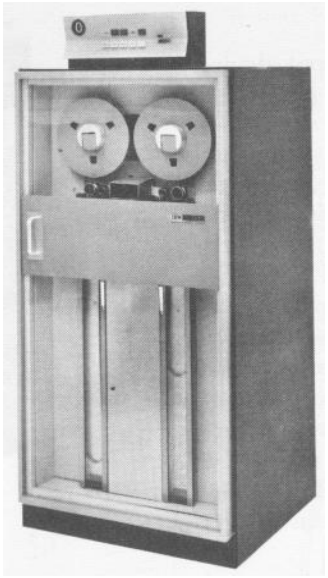


Cast operation – The resultant width is *N*

An ALU operation. *op* can be one of the following:

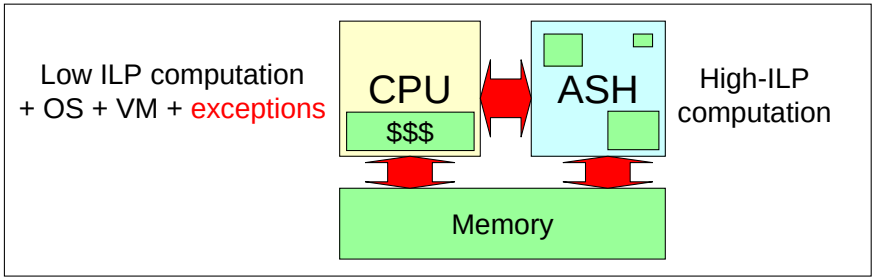
+	Addition
-	Subtraction/Negation
*	Multiplication
muluh	High bits of multiplication
/	Division
%	Remainder
<<	Left Shift
>>	Logical/Arithmetic Right Shift
==	Is equal to?
!=	Is not equal to?
<	Less than?
<=	Less than equal to?
!	Logical Not
&&	Logical AND
||	Logical OR
~	Bitwise complement
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR

Backup Slides

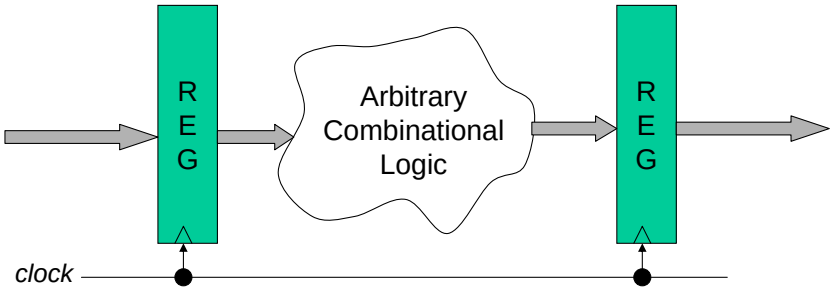


Exceptions

- Strictly speaking, C has no exceptions
- In practice hard to accommodate exceptions in hardware implementations
- An advantage of software flexibility: PC is single point of execution control

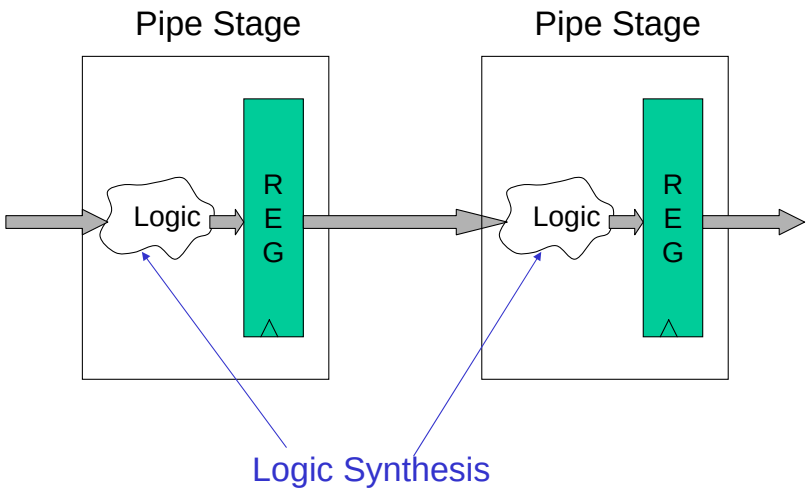


Synchronous Synthesis



- Logic Minimization occurs on the logic between registers
- Combinational Logic must not contains cycles

Asynchronous Synthesis



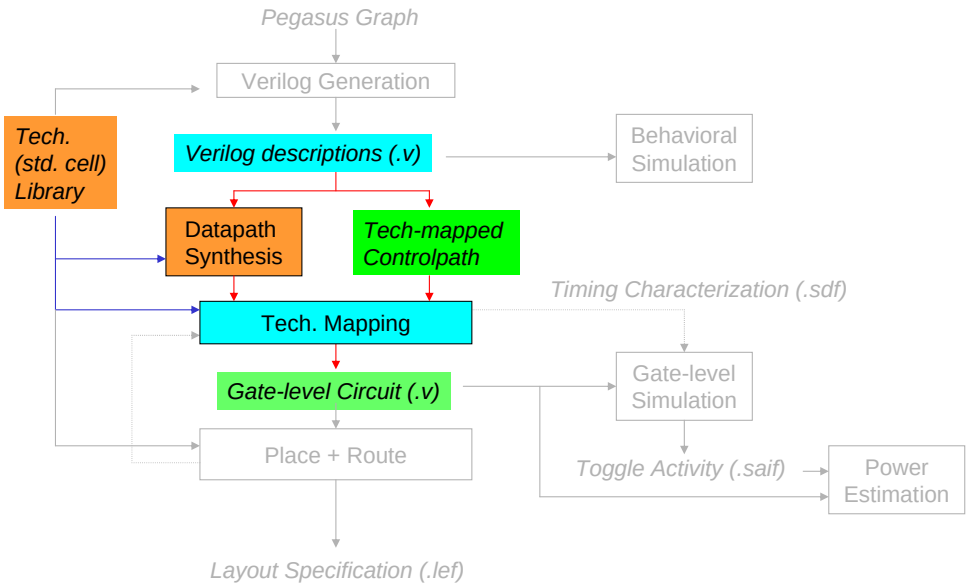
Synthesis with Synopsys

- Fully adapted Synopsys DC for our circuits
- Synchronous synthesis
  - Logic Minimization
  - Circuit Inferrence (eg. FSM, registers)
  - Tech. Mapping

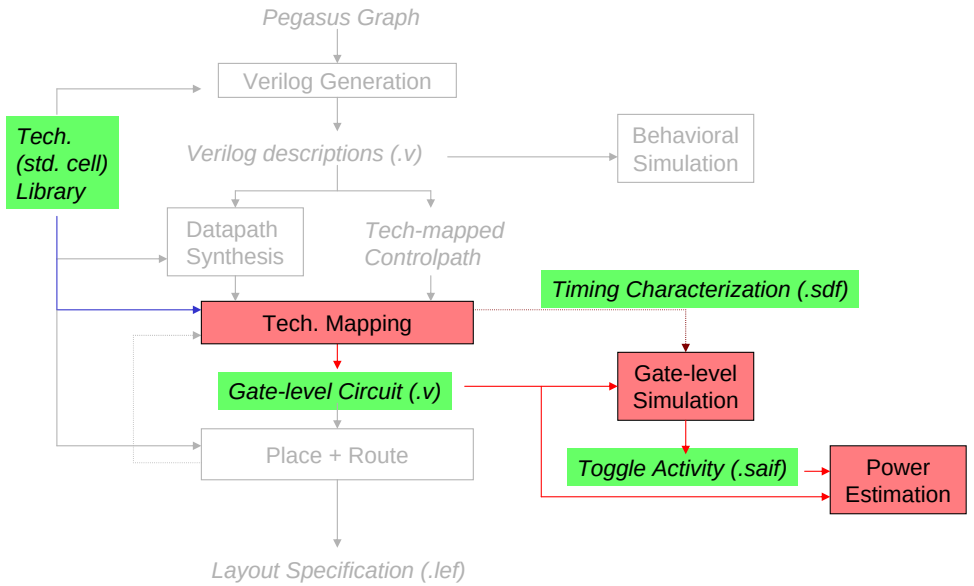
Synthesis with Synopsys

- Fully adapted Synopsys DC for our circuits
- Asynchronous synthesis
  - ✓Logic Minimization -- Synopsys DC
  - ✗Circuit Inferrence (eg. FSM, registers)
  - ✓Tech. Mapping -- Synopsys DC + CAB

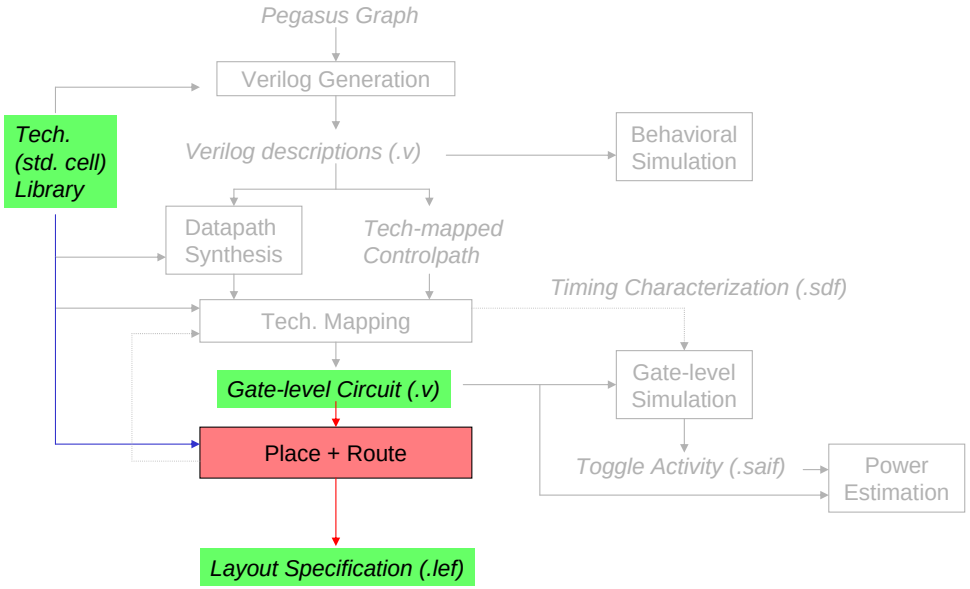
Design Flow



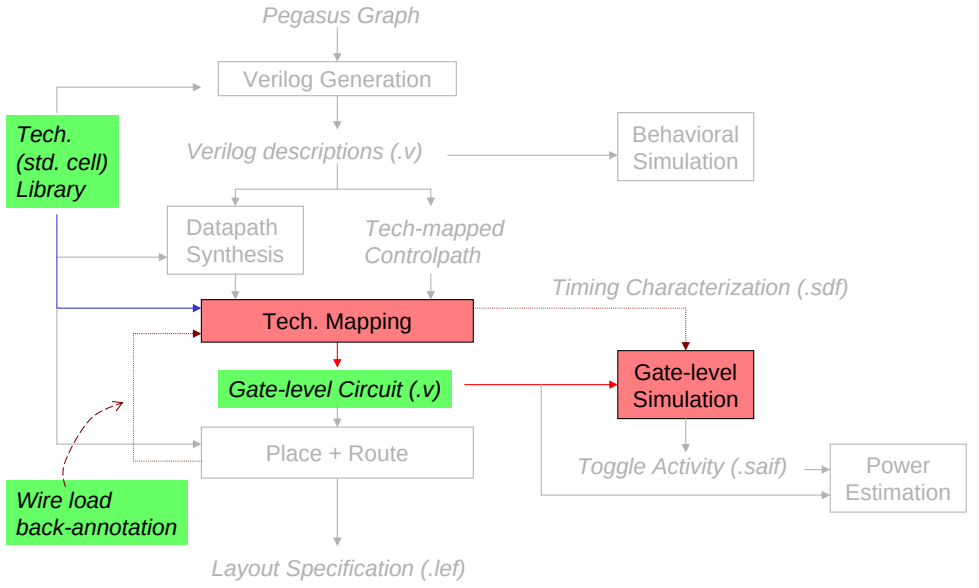
Design Flow



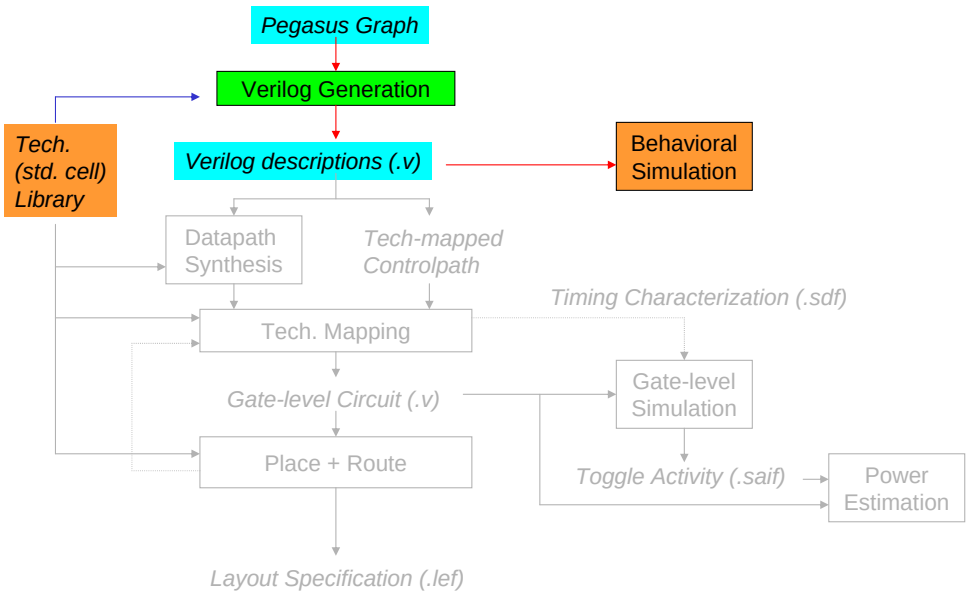
Design Flow



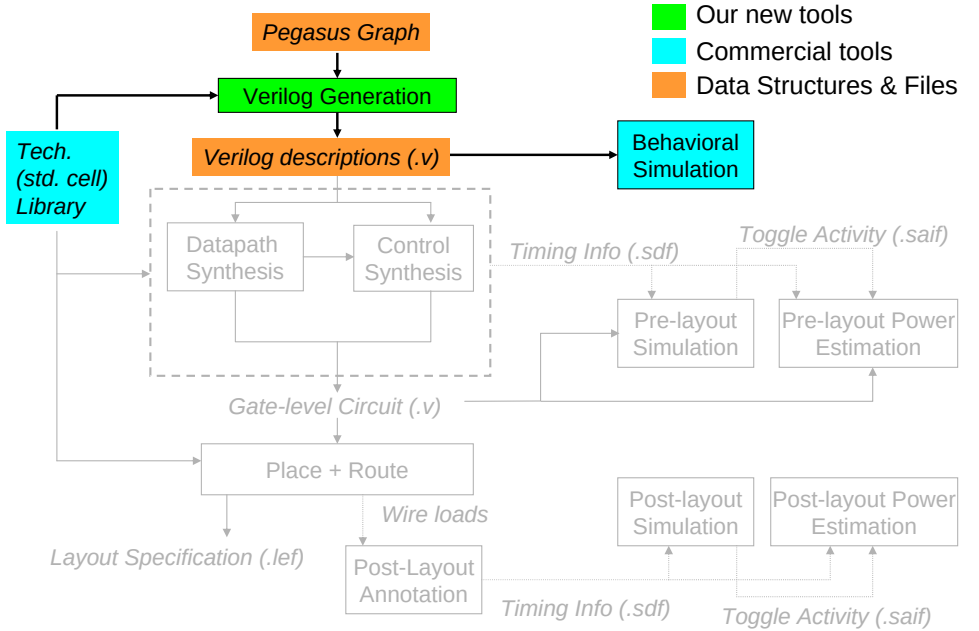
Design Flow



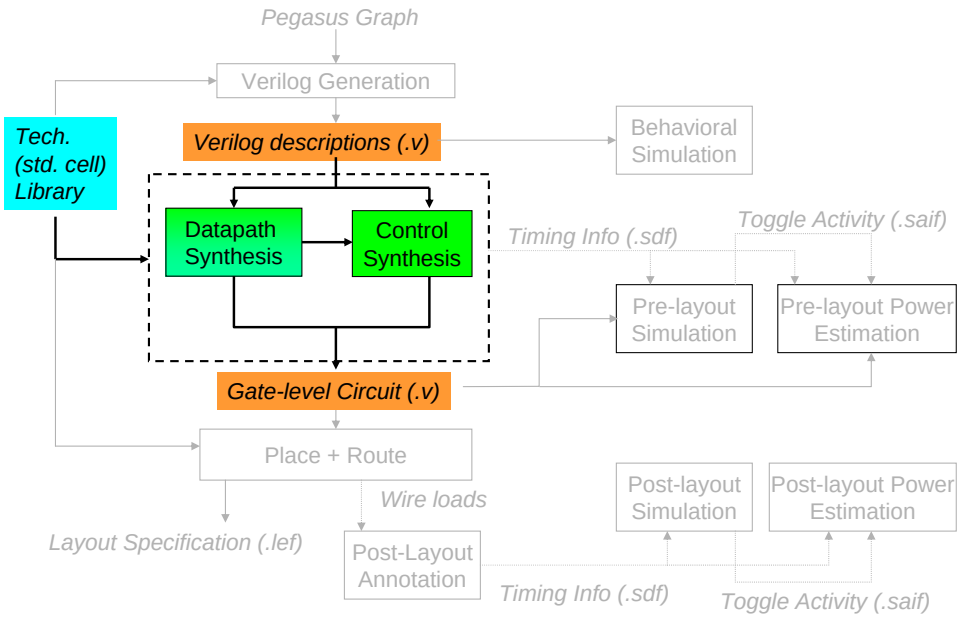
Design Flow



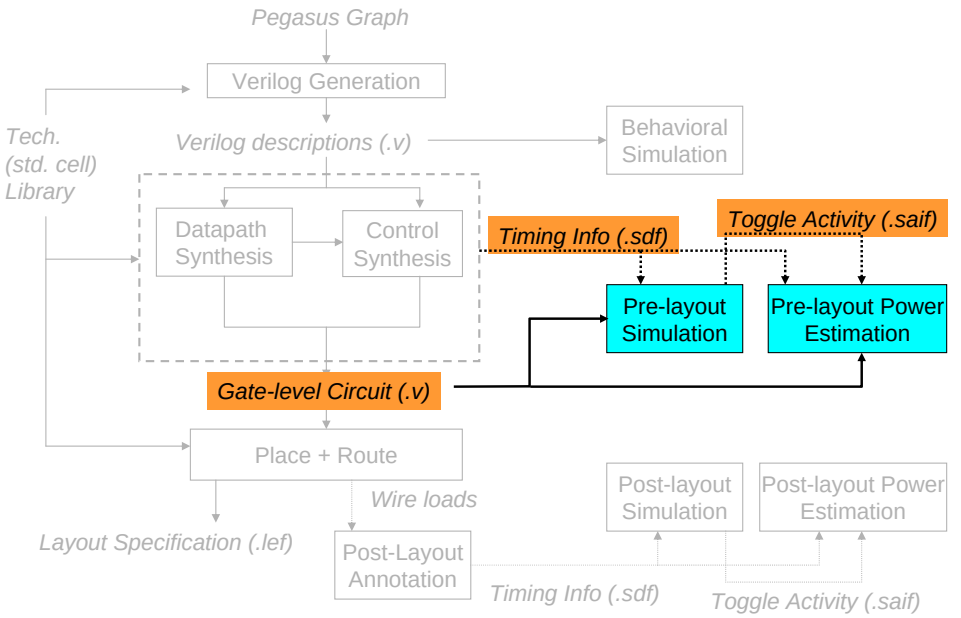
Design Flow: Behavioral Synthesis



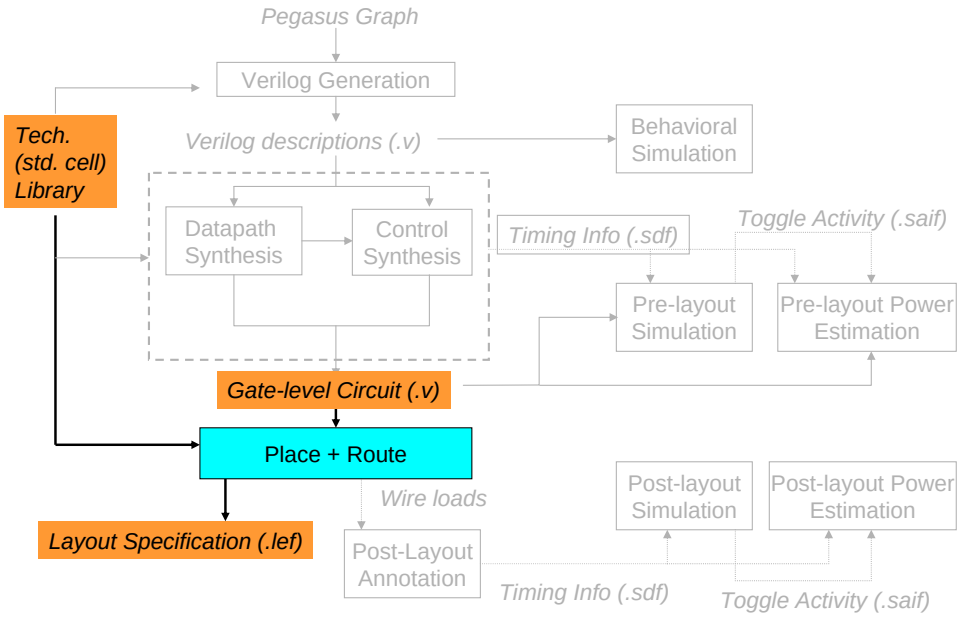
Design Flow: Synthesis & Tech-Mapping



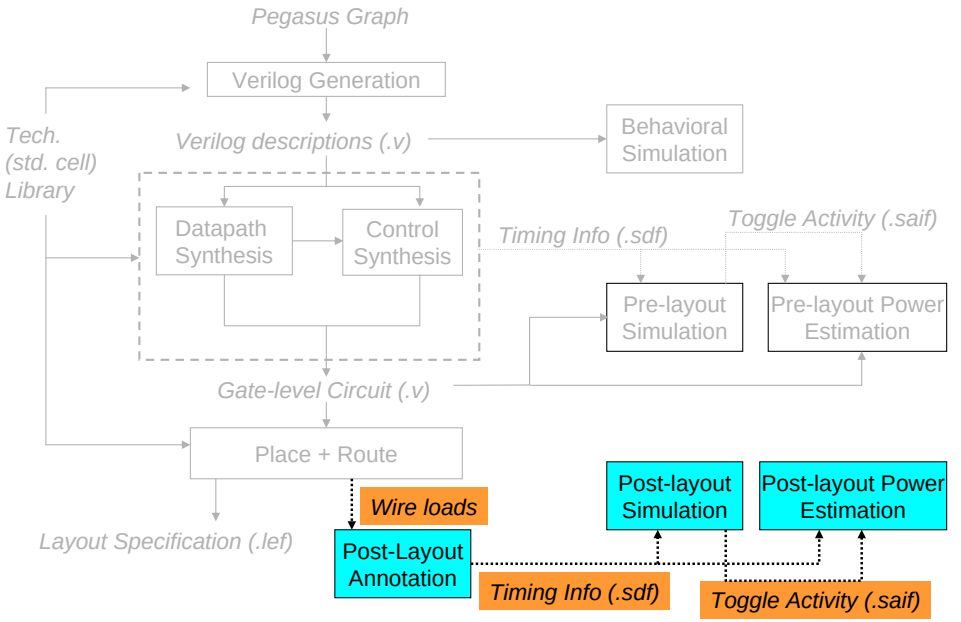
Design Flow: Pre-Layout Simulation



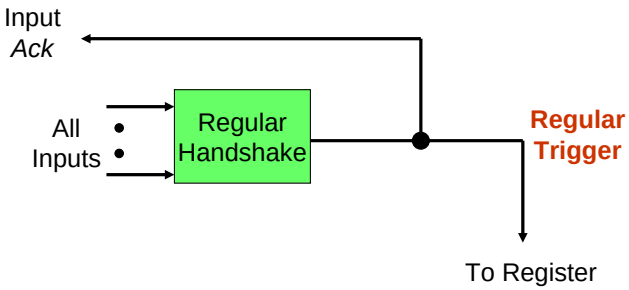
Design Flow: Layout



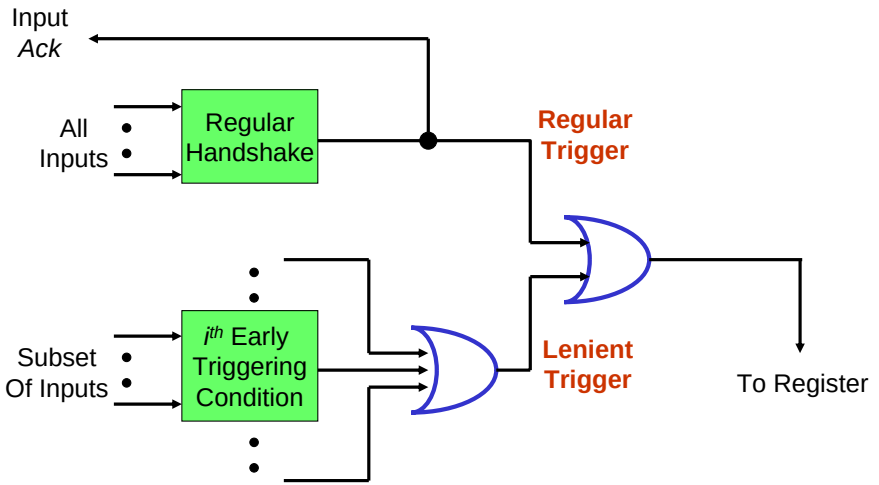
Design Flow: Post-Layout Simulation



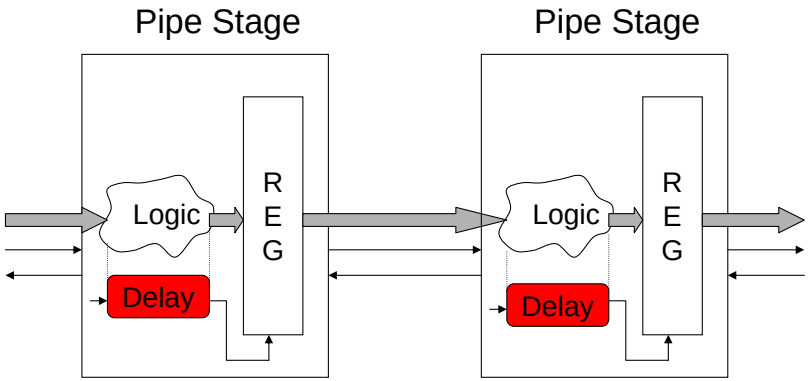
Lenient Handshake



Lenient Handshake



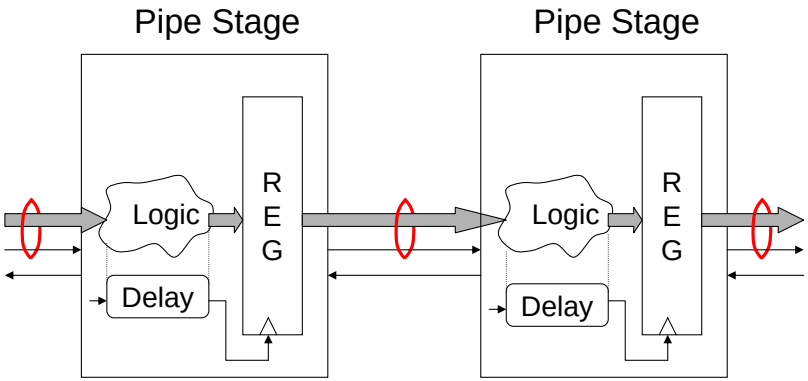
Timing Dependencies: Matched Delay



Q) Does the delay match the logic delay?

A) Limited fanout – easier to estimate delay

Timing Dependencies: Bundled Data

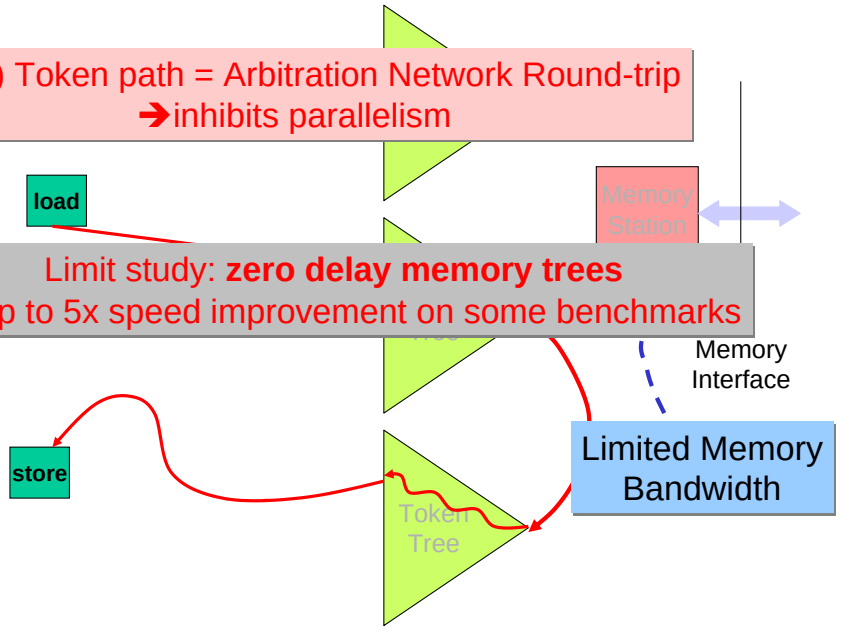


Q) Is the bundling constraint satisfied?

A) Provide hints to guide the layout tool

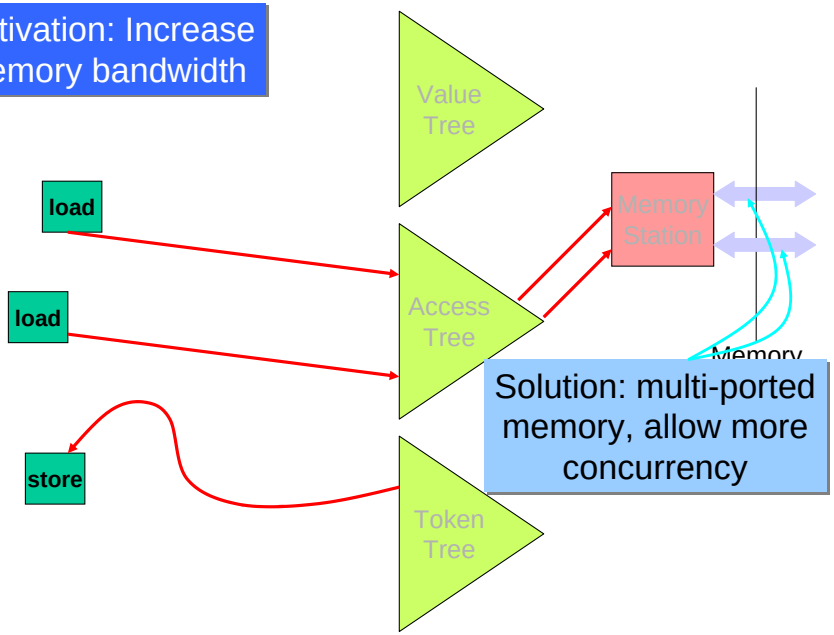
Weaknesses of Design

(Long) Token path = Arbitration Network Round-trip
➔inhibits parallelism



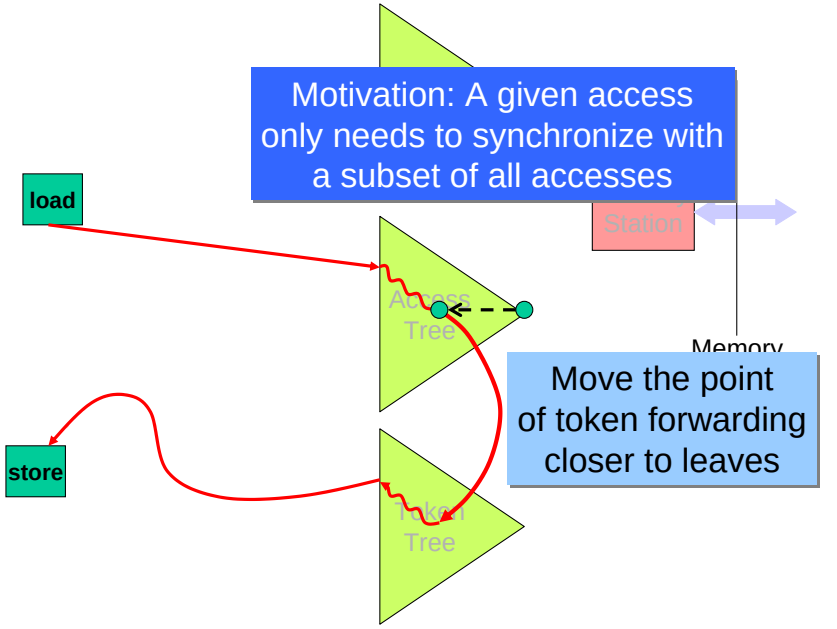
Alternatives: Multi-ported Memory

Motivation: Increase memory bandwidth



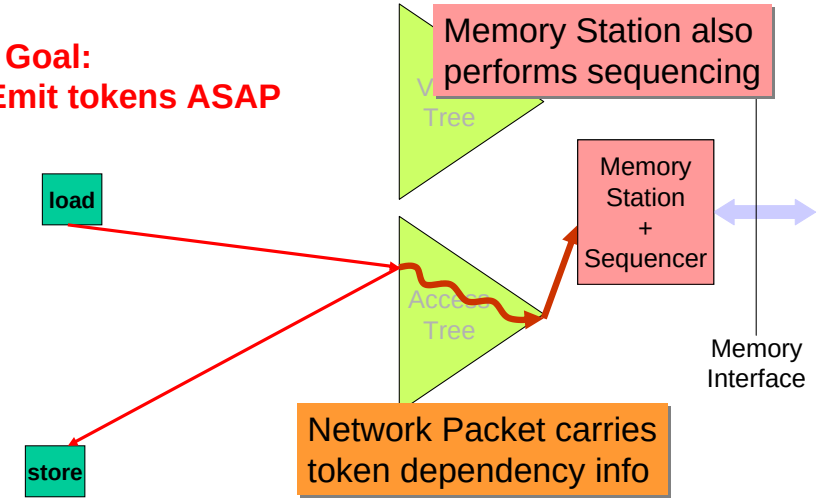
Alternatives: Shorten Token Path

Motivation: A given access only needs to synchronize with a subset of all accesses

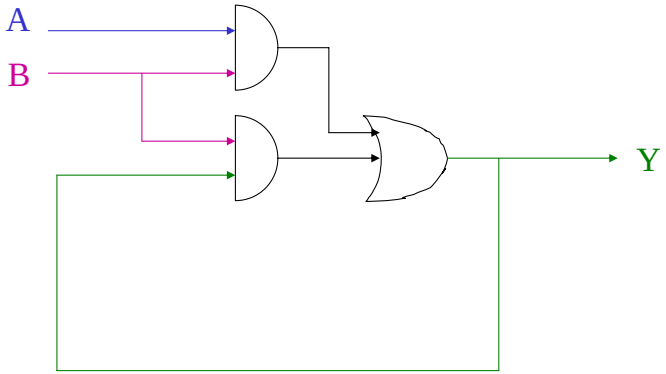


Alternatives: New Protocol

Design Goal:
Emit tokens ASAP



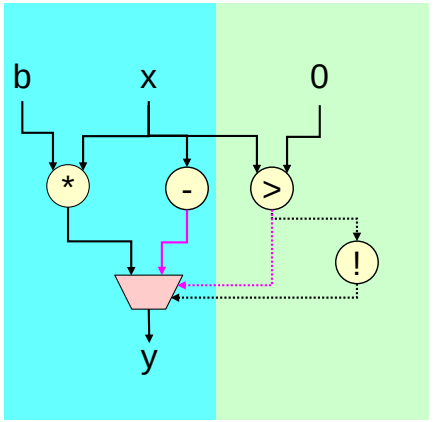
Assymetric C-Element



Back

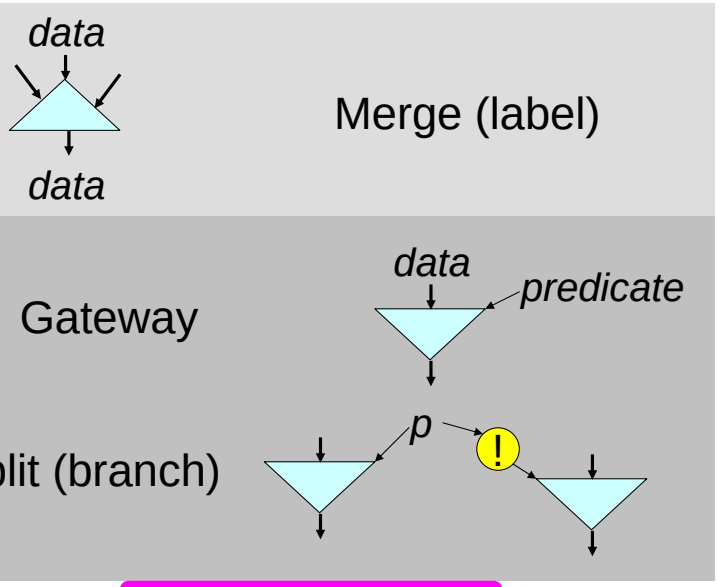
MUX: Forward Branches

```
if (x > 0)
        y = -x;
else
        y = b*x;
```



C → CFG → ∑ acyclic → dataflow → circuits

Control Flow ⇒ Data Flow



C → CFG → ∑ acyclic → dataflow → circuits

Simulate the Verilog (on ECE)

- Transfer files to ECE
  - $ scp demo_squares/sim_ver <username>@delta.ece:squares
- Connect to ECE
  - $ ssh –X delta.ece.cmu.edu –l <*your username*>
- Simulate using Verilog-XL
  - $ bash
  - $ cd squares
  - $ source /afs/cs/project/phoenix/async_tutorial/vxl_setup.sh
  - $ verilog +gui *.v
  - – Type “$finish;” from within the simulator to exit
- Note and save the simulation time
  - $ grep “donetoken = 1” verilog.log
  - $ cp verilog.log squares1.log
- Return to CS
  - $ exit // from bash shell
  - $ exit // from ECE

Simulate the Verilog (on ECE)

- Simulate using Verilog-XL
  - $ bash
  - $ cd squares
  - $ source /afs/cs/project/phoenix/async_tutorial/vxl_setup.sh
  - $ verilog +gui *.v
  - $ Type “$finish;” from within the simulator to exit
- Compare the simulation times
  - $ grep “donetoken = 1” verilog.log squares1.log
- Return to CS
  - $ exit // from bash shell
  - $ exit // from ECE