Compiling C to Asynchronous Hardware

Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, Seth Goldstein
{mihai, girish, tibi, seth}@cs.cmu.edu

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

Imagine: Computers that
• Small enough to fit inside cells
• Cheap enough to be disposable
• Dense enough to embed a supercomputer
• Smart enough to assemble themselves

Computers from atomic scale components

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

As we scale down:

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

Requiring:

- Defect tolerant architectures
- Higher level specification
- Universal substrate

Technical Challenges

Delay

<table>
<thead>
<tr>
<th>gate</th>
<th>5ps</th>
</tr>
</thead>
<tbody>
<tr>
<td>wire</td>
<td>10ps</td>
</tr>
</tbody>
</table>

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

Complexity Challenges

+ Testing
+ Verification

+ Testing
+ Verification

Mask defect of 3 atoms
→ chip defect!

$\text{\$\$\$\$\$\$ Challenges}$
Manufacturing Paradigm Shift Required

Today
• Reliable Systems from reliable components

• Functionality invested at time of manufacture

• Behavior remains same as features scales down

Future
• Reliable Systems from unreliable components

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

Universal gates
and/or
storage elements

Programmable Switches

Interconnection network
DFT tool flow

Fabric → Netlist

Circuit → Place-and-Route

Tester

Defect Map

Defect Aware Place-and-Route

“Hard” Config.

“Soft” Config.

Async '04 © 2002-4 Carnegie Mellon University

Reconfigurability & DFT

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

This Tutorial

Reconfigurable Computing

General-Purpose

Custom Hardware

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

int func(int*, int*)

编译器

逻辑块

近端

您必须存储配置！

Advantages of Reconfigurable

- Flexibility of a processor
- Performance of custom hardware

您必须存储配置！
Heart of an FPGA

Switch controlled by a 1-bit RAM cell

Universal gate = RAM

The cost of the FPGA:
- Increased Area-Delay Product

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

Control, configuration decompression, & defect mapping seed

The Molecular Electronics Advantage:
- Low design cost
- Low power
- Inexpensive manufacturing
- Reconfigurable
- Defect and fault tolerance

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

You are here
Spanning 10-orders of Magnitude

1 Program

Compilers

Phoenix

Theory

Architecture

10 Billion Gates

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

C Program

Circuits

Memory partitioning

Interconnection net
Circuits From Compilers

1. Program
2. Split-phase Abstract Machines
3. Configurations placed independently
4. Placement on chip

Certifying Circuits!

Computations & local storage
Unknown latency ops

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 ...

Current Usage Methodology

C Application

HW/SW partitioning

selected functions
selected functions

This toolflow

C compiler

This toolflow

“coprocessor” custom/reconf

CPU

“coprocessor” custom/reconf

memory

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

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 mm²
- **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
**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

**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

**Tutorial Outline**

- Part I: CASH
  - C
  - Front-end
  - Optimizer
  - Back-end
- Part II: CAB
  - Structured verilog
  - Synthesis + P/R
  - Layout
  - Std. cell library

**ASH Area**

- Mem access
- Datapath

**Implementation Overview**

- P4: 217

**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
Part I
From C to Dataflow

Computation = Dataflow

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

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

Circuits

Basic Computation

Asynchronous Computation
Compilation Outline

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

Hyperblocks

Predication

Speculation

ops w/ side-effects (load, store, call)
**Example Conditional**

\[
\text{if } (x > 0) \\
\quad y = -x; \\
\text{else} \\
\quad y = b^*x; \\
\text{... } = y;
\]

\[
p = (x > 0) \\
\text{if } (p) \\
\quad y = -x; \\
\text{else} \\
\quad y = b^*x; \\
\text{... } = y;
\]

\[
p = (x > 0); \\
y_1 = -x; \\
y_2 = b^*x; \\
y = \text{mux}(p, \neg p, y_1, y_2); \\
\text{... } = y;
\]

**MUX = Single-Assignment**

\[
\text{if } (x > 0) \\
\quad y = -x; \\
\text{else} \\
\quad y = b^*x;
\]

**Critical Paths**

\[
\text{if } (x > 0) \\
\quad y = -x; \\
\text{else} \\
\quad y = b^*x;
\]

**Lenient (Early) Evaluation**

\[
\text{if } (x > 0) \\
\quad y = -x; \\
\text{else} \\
\quad y = b^*x;
\]

\[\text{Solves the problem of unbalanced paths}\]
Stitching Hyperblocks

Loops: Control Flow $\Rightarrow$ Data Flow

Control Flow $\Rightarrow$ Data Flow

Loops

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

**Visualization**

Event-driven simulation

- CASH
- gcc
- C program
- picture.dot
- images
- Visualization script
- dot
- circuit animation
- dynamic critical path
- C + runtime lib
- low-level

**Visualization Options**

- event trace
- reference input
- output
- diff
- reference output

**Visualization**

- Circuit picture
- Execution trace
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: tutX (X = number on the machine CAD_PCX)
   • 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

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

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

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

Pipelining

```
step=1
```

Pipelining

```
step=2
```

Pipelining

```
step=3
```
Pipelining

step=4

Pipelining

step=5

Pipelining

step=6

Pipelining

step=7
Predicate \( \text{ack} \) edge is on the critical path.

Pipeline balancing

\[ i \leq 100 \]

\[ \sum \text{sum}'s \ loop \]

\[ \text{decoupling FIFO} \]

Step = 7

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

Predication and Side-Effects

Memory Access

no speculation

Load

sequencing of side-effects
to memory

LD

ST

Monolithic Memory

local communication
global structures

Future work: fragment this!
Procedure calls

Arguments

Procedure P

caller callee

ret

result caller

Recursion

save live values

recursive call

restore live values

stack

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

Part II

From Dataflow to Asynchronous Circuits

C → CFG → Σ acyclic → dataflow → 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
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 Graph

Verilog Generation

Verilog descriptions (.v)

Synthesis

Gate-level Circuit (.v)

Place + Route

Layout Specification (.lef)

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

4-phase Handshake

Data
Req
Ack

Stage Logic

Output Register Logic

Delay

Req

Ack

Input Bundle

Output Bundle

a'la Sutherland's Micropipelines

Pipe Stage Architecture

Delay

Stage Logic

Output Register Logic

Input Bundle

Output Bundle

Pipe Stage Architecture

Delay

Stage Logic

Output Register Logic

Input Bundle

Output Bundle

Pipe Stage Architecture

Delay

Stage Logic

Output Register Logic

Input Bundle

Output Bundle

Pipe Stage Architecture

Delay

Stage Logic

Output Register Logic

Input Bundle

Output Bundle
Completion Detection: 2-bit Register

Input 2

Enable

AC

Done

Output 2

Completion Detection: 2-bit Register

Input 2

Enable

AC

Done

Output 2

((Input == Output) && Enable)?

AC

Completion Detection: 2-bit Register

Input 2

Enable

AC

Done

Output 2

Pipe Stage Architecture

Input Bundle

Stage Logic

Delay

Req

Ack

Output Bundle

Req

Ack

CD
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 –I 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

Leniency Overhead

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 --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
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

Pegasus Graph

- Verilog Generation
  - Tech. (std. cell) Library
  - Verilog descriptions (.v)
- Synthesis
  - Gate-level Circuit (.v)
- Place + Route
  - Layout Specification (.lef)

Our new tools
Commercial tools

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)

### Delay Matching

\[ \Delta = D_{\text{logic}} - (D_{\text{CD}} + D_{\text{HS}}) + \text{slack} \]

- \( D_{\text{logic}} \) 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

- **Yes; (at the previous stage’s output due to CD)**
- **Between stages, the onus is on the layout tool**
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
  
  $\text{grep } \text{"donetoken = 1" transcript pre_layout_sim.time}$

Compiler Optimizations

- Check compiler options
  
  $\text{c2verilog.pl } -h$
  
  -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 $<\text{bench}>$, and function is $<\text{func}>$

- Compile:
  
  $\text{c2verilog.pl } [\text{options}] <\text{bench}> <\text{func}>$

- If you specify –p $<\text{file}>$ above, then you can view the critical path:
  
  $\text{dotty } <\text{file}>$

  - Use the Dotty hand-out to interpret the graph

- If you specify –v $<\text{file}>$ above, then you generate verilog

- Verilog Simulation
  
  $\text{cd } <\text{bench}>/\text{sim_ver}$
  
  $\text{verilog +gui *.v}$
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

<table>
<thead>
<tr>
<th>Feature</th>
<th>Advantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>No interpretation</td>
<td>Energy efficiency, speed</td>
</tr>
<tr>
<td>Spatial layout</td>
<td>Short wires, no contention</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>Low power, scalable, tolerant to variation</td>
</tr>
<tr>
<td>Distributed</td>
<td>No global signals</td>
</tr>
<tr>
<td>Automatic compilation</td>
<td>Design productivity</td>
</tr>
</tbody>
</table>

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 cm²
  - 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

<table>
<thead>
<tr>
<th>File Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>circ_io_wrapper.v</td>
<td>Top-level testbench</td>
</tr>
<tr>
<td>mem_wrapper.v</td>
<td>Memory Emulator</td>
</tr>
<tr>
<td>circ_df.v</td>
<td>Structural Verilog circuit</td>
</tr>
<tr>
<td>circ_mem.v</td>
<td>Memory Access Tree</td>
</tr>
<tr>
<td>pegasuslib_gen.v</td>
<td>Pegasus Macro libraries</td>
</tr>
<tr>
<td>pegasusmem_gen.v</td>
<td></td>
</tr>
<tr>
<td>gates_bhv.v</td>
<td>Standard Cell library abstraction</td>
</tr>
<tr>
<td>macros_bhv.v</td>
<td></td>
</tr>
</tbody>
</table>

Understanding the Dotty Graph

Dotty Key Bindings

- ‘z’    Zoom out
- ‘Z’    Zoom in
- ‘r’    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)

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.

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

- Addition
- Subtraction/Negation
- Multiplication
- 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
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
Synthesis with Synopsys

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

Design Flow

1. Pegasus Graph
2. Verilog Generation
3. Verilog descriptions (.v)
4. Tech. (std. cell) Library
5. Datapath Synthesis
6. Tech-mapped Controlpath
7. Tech. Mapping
8. Gate-level Circuit (.v)
9. Place + Route
10. Layout Specification (.lef)
11. Behavioral Simulation
12. Gate-level Simulation
13. Power Estimation
14. Toggle Activity (.saif)
15. Timing Characterization (.sdf)
Lenient Handshake

All Inputs → Regular Handshake → Regular Trigger → To Register

Timing Dependencies: Matched Delay

Pipe Stage Logic → REG Delay → Pipe Stage Logic → REG Delay

Q) Does the delay match the logic delay?
A) Limited fanout – easier to estimate delay

Timing Dependencies: Bundled Data

Pipe Stage Logic → REG Delay → Pipe Stage Logic → REG Delay

Q) Is the bundling constraint satisfied?
A) Provide hints to guide the layout tool
Weaknesses of Design

(Long) Token path = Arbitration Network Round-trip
\[\Rightarrow\] inhibits parallelism

Limit study: zero delay memory trees
\[\Rightarrow\] up to 5x speed improvement on some benchmarks

Alternatives: Multi-ported Memory

Motivation: Increase memory bandwidth

Solution: multi-ported memory, allow more concurrency

Alternatives: Shorten Token Path

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

Move the point of token forwarding closer to leaves

Alternatives: New Protocol

Design Goal:
Emit tokens ASAP

Network Packet carries token dependency info

Memory Station also performs sequencing
**Assymetric C-Element**

A

B

Y

**MUX: Forward Branches**

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

**Control Flow ⇒ Data Flow**

data

Merge (label)

data

Gateway

data

predicate

Split (branch)

p

C → CFG → ∑ acyclic → dataflow → circuits

**Simulate the Verilog (on ECE)**

- Transfer files to ECE
  
  ```bash
  $ scp demo_squares/sim_ver <username>@delta.ece:squares
  ```

- Connect to ECE
  
  ```bash
  $ ssh -X delta.ece.cmu.edu -l <your username>
  ```

- Simulate using Verilog-XL
  
  ```bash
  $ 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
  
  ```bash
  $ grep "donetoken = 1" verilog.log
  $ cp verilog.log squares1.log
  ```

- Return to CS
  
  ```bash
  $ 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 "dgen = 1" verilog.log squares1.log

- Return to CS
  $ exit // from bash shell
  $ exit // from ECE