AHIR

A competitive pathway from high-level programs to hardware specifications.

Sameer D. Sahasrabuddhe
Advisors:
Prof. Kavi Arya (CSE)
Prof. Madhav P. Desai (EE)

IIT Bombay

August 24th, 2009

High-level Synthesis

Translating a program to hardware.

- Reduce dependence on hardware design skills
  - Programs specify the behaviour of the circuit.
  - “Even programmers” can create hardware.
- Simplify verification
  - Generated RTL is correct by construction.
  - Verify the program instead of the generated RTL.
- Faster design flow.
- Design large systems within manageable design costs.

A *competitive* high-level synthesis flow

- Independent of the input programming language.
- Produce hardware that is provably correct.
- Produce efficient hardware.
- Scale to very large systems.

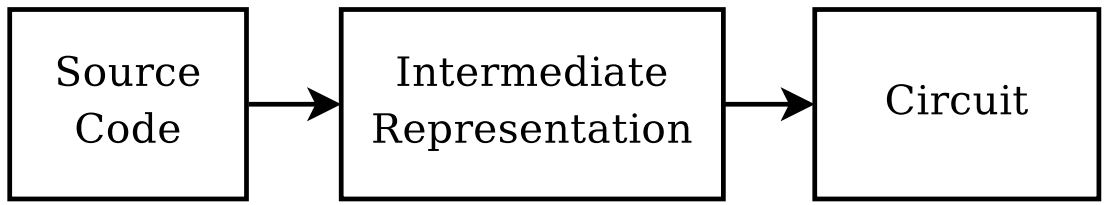
Problem

Translating behaviour to structure.

- The input is an algorithm.
- The output is a hardware description.

Solution

Introduce an intermediate representation.



- Hides low-level details.
  - Target for compiling the high-level program.
- Exposes information relevant to synthesis.
  - A structure that implements the behaviour.

Control Data Flow Graph (CDFG)

- Traditionally used as an IR for high-level synthesis.
- Control and data are expressed in a single graph.

Decoupled control and data.

- Separately implement the two components.
  - Scalable to large programs.
- Separation of *resource* from *computation*.
  - New optimisations that work across modules.

AHIR: A Hardware Intermediate Representation

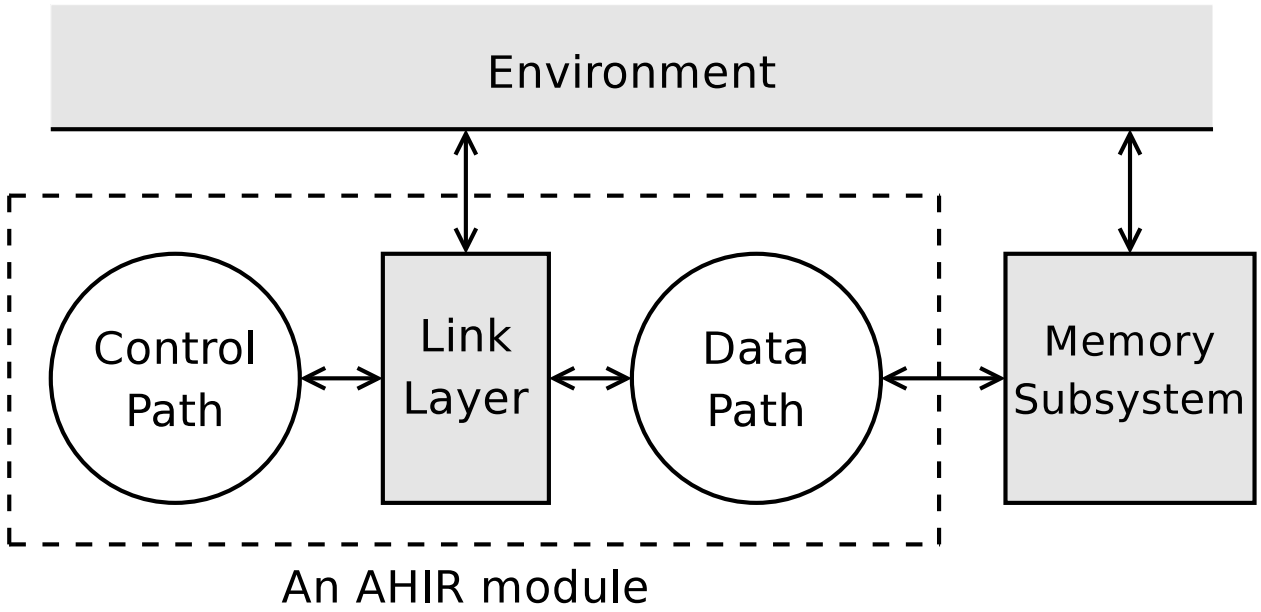
- Orthogonal factorisation of the program
  - Control $\times$ Data $\times$ Storage
- Special petri-net class that supports scalable analysis.
- Request–Acknowledge handshakes.
  - One-sided delay constraints.
- Independent memory subsystem.

High-level Synthesis with AHIR

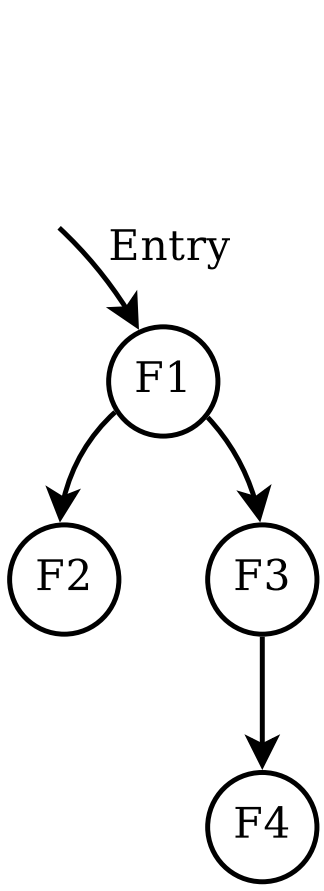
- CDFG as an intermediate step.
  - The generated AHIR is equivalent to the input CDFG.
- Supports scalable optimisation.
  - Arbiterless sharing.
- Routine translation to hardware circuits.
  - Demonstrated on a diverse set of applications.
  - Competitive with embedded processors.
  - Potentially get very close to hand-crafted circuits.

Outline

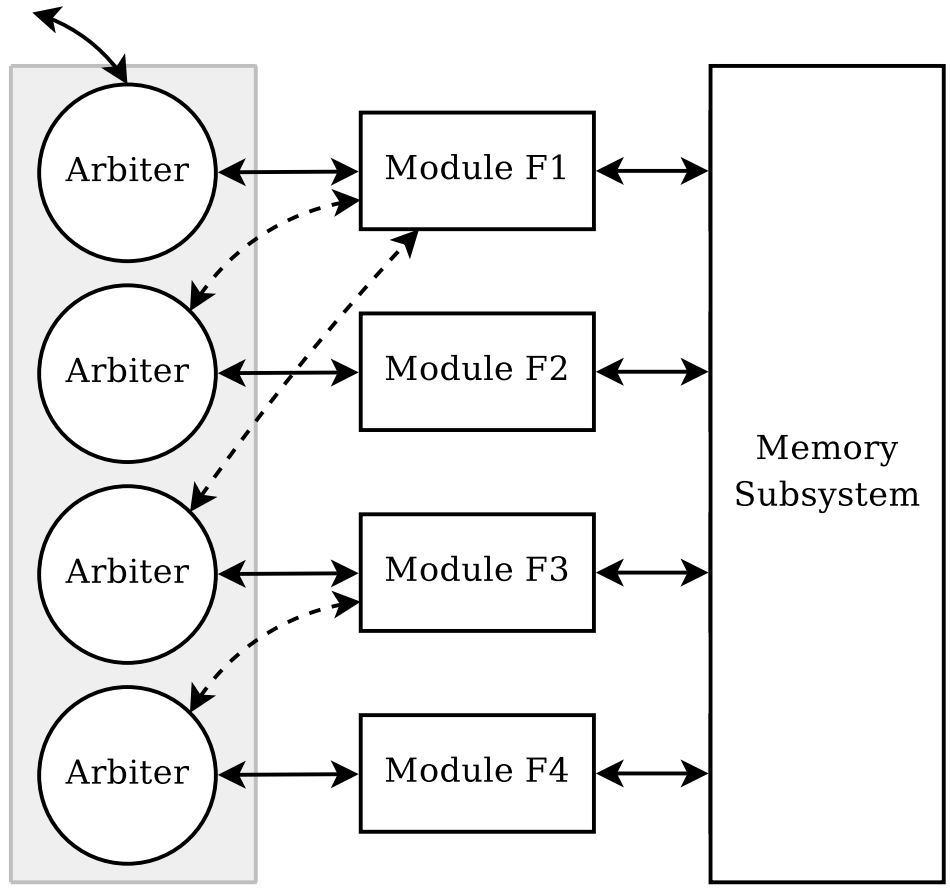
An AHIR Module



An AHIR system

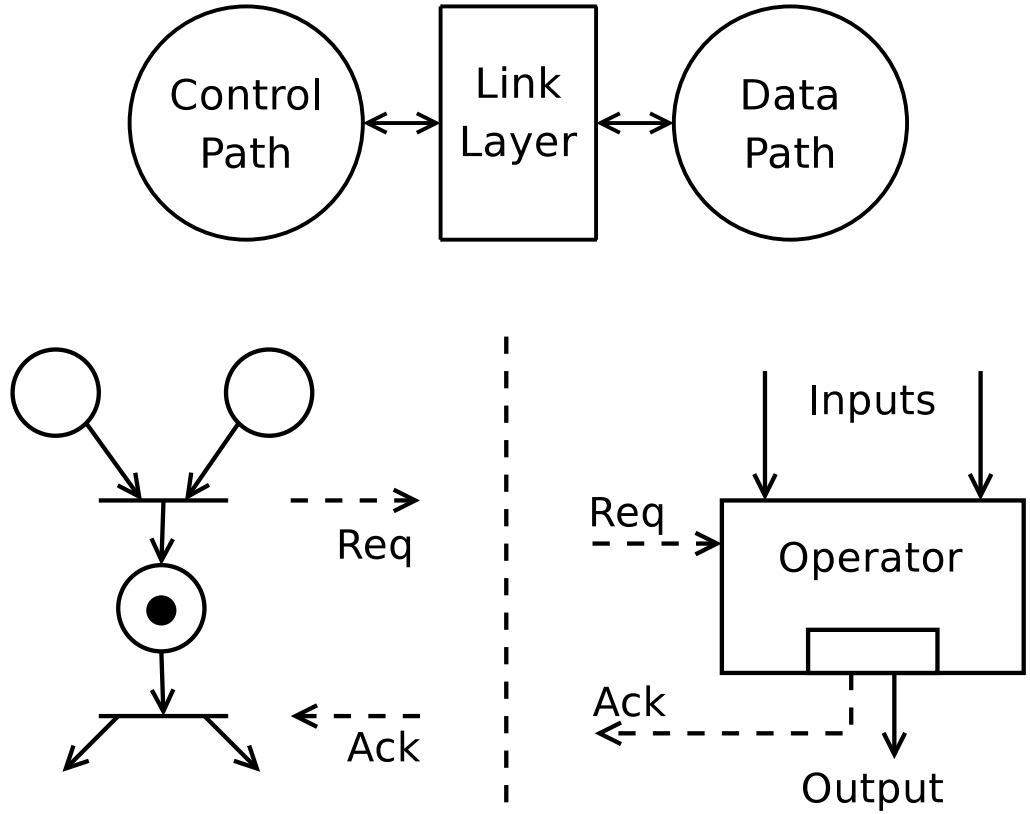


Call-graph

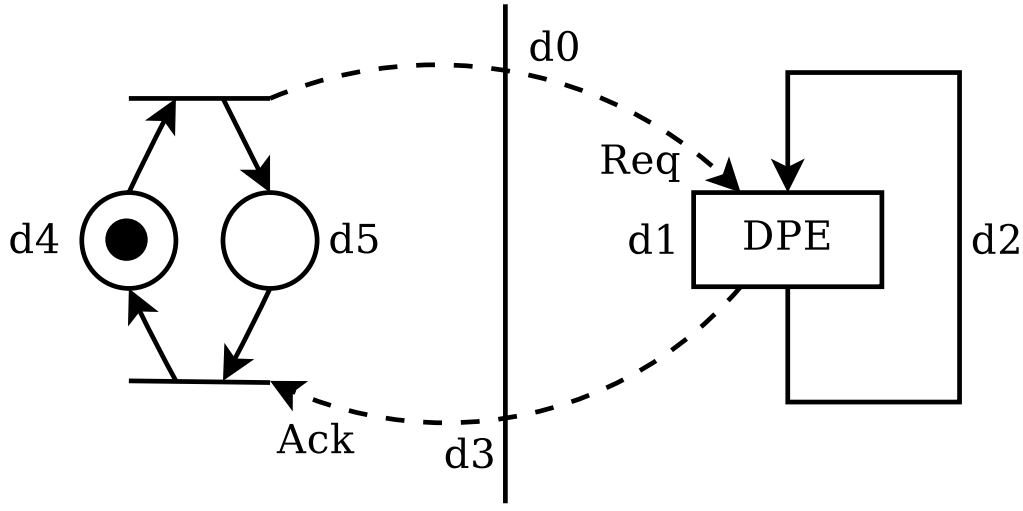


Modules

Handshakes


Control Path
Link Layer
Data Path
Req
Ack
Inputs
Req
Operator
Ack
Output

Timing constraints



$$d_5 \leq d_0 + d_1 + d_3$$

$$d_2 \leq d_3 + d_4 + d_0$$

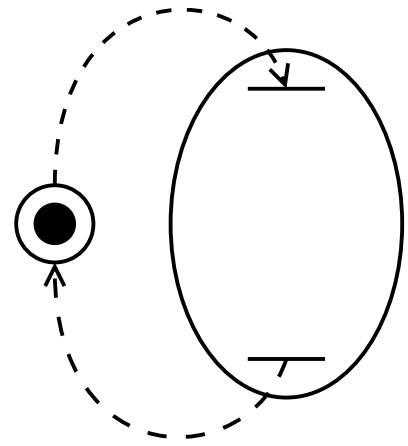
Control-path as a petri-net

- A general petri-net can be arbitrarily complex.
- We need a subclass that is:
  - live and safe
  - easy to analyse
  - not restrictive
- Type-2 petri-net

Type-2 petri-nets

Token Preserving Region: a connected subgraph of a petri-net:

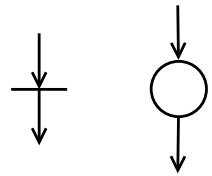
- Has a unique entry and a unique exit.
- No place marked in the initial marking.
- Connecting a marked place results in a live and safe petri-net.



Type-1 Petri-net: Created from a TPR.
Type-2 Petri-net: A subset where the TPR is constructed using specific rules.

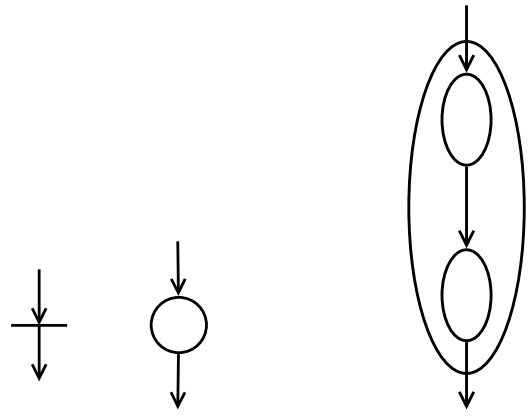
Type-2 production rules

Type-2 production rules

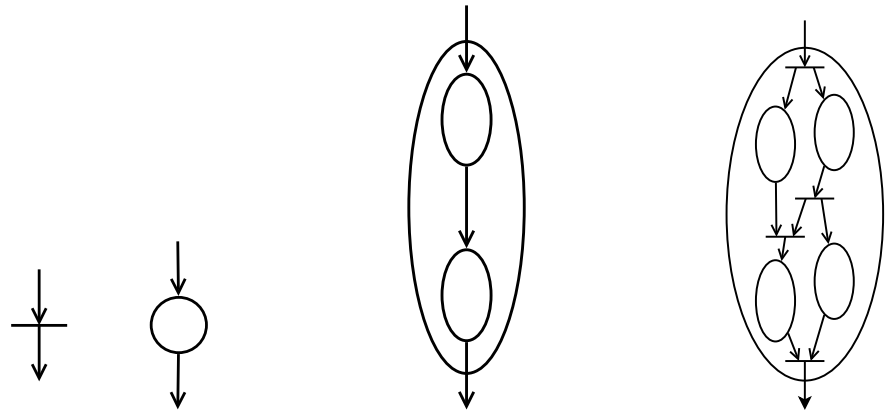


Primitive

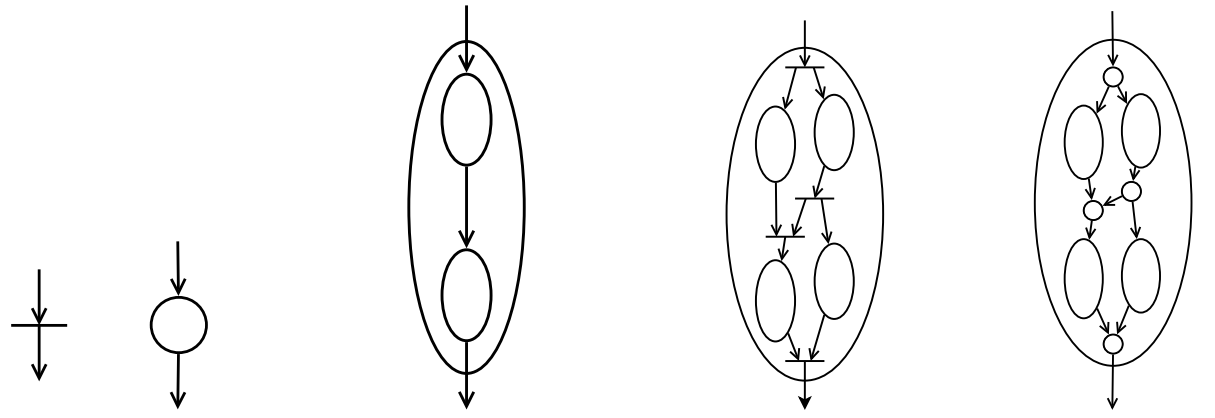
Type-2 production rules



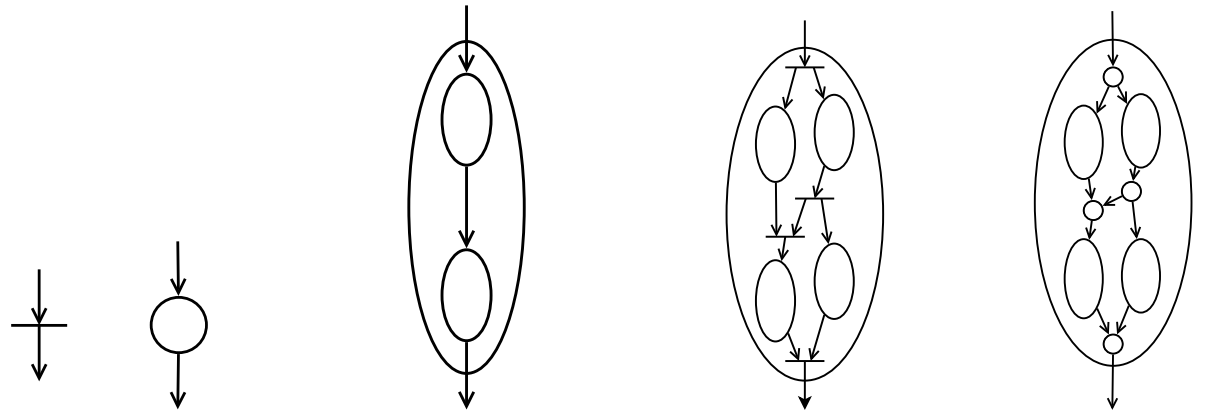
Type-2 production rules

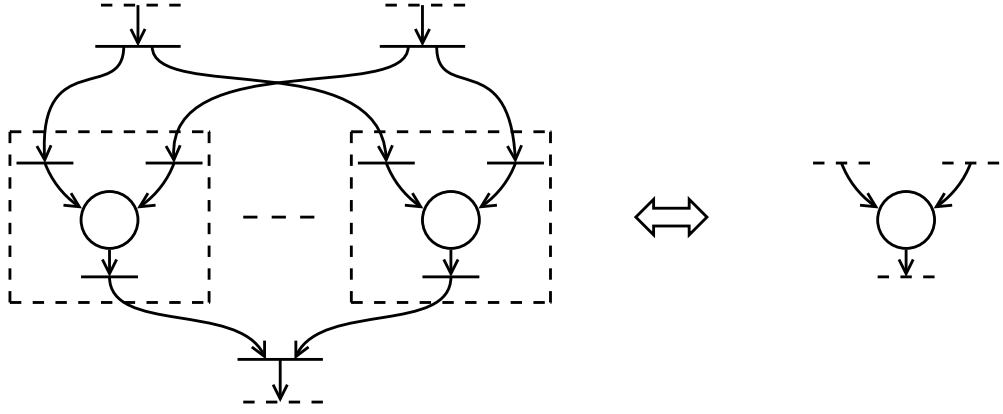


Type-2 production rules



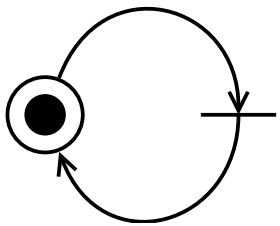
Type-2 production rules



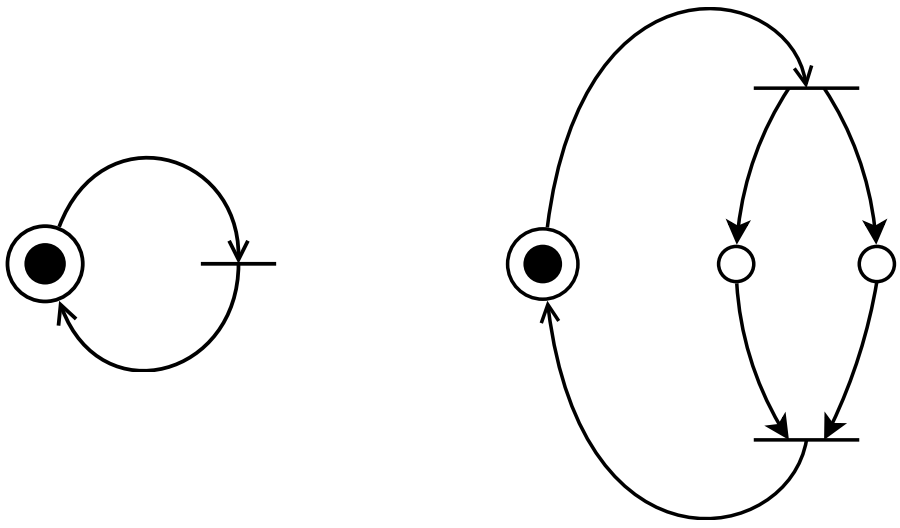


Parallel merges

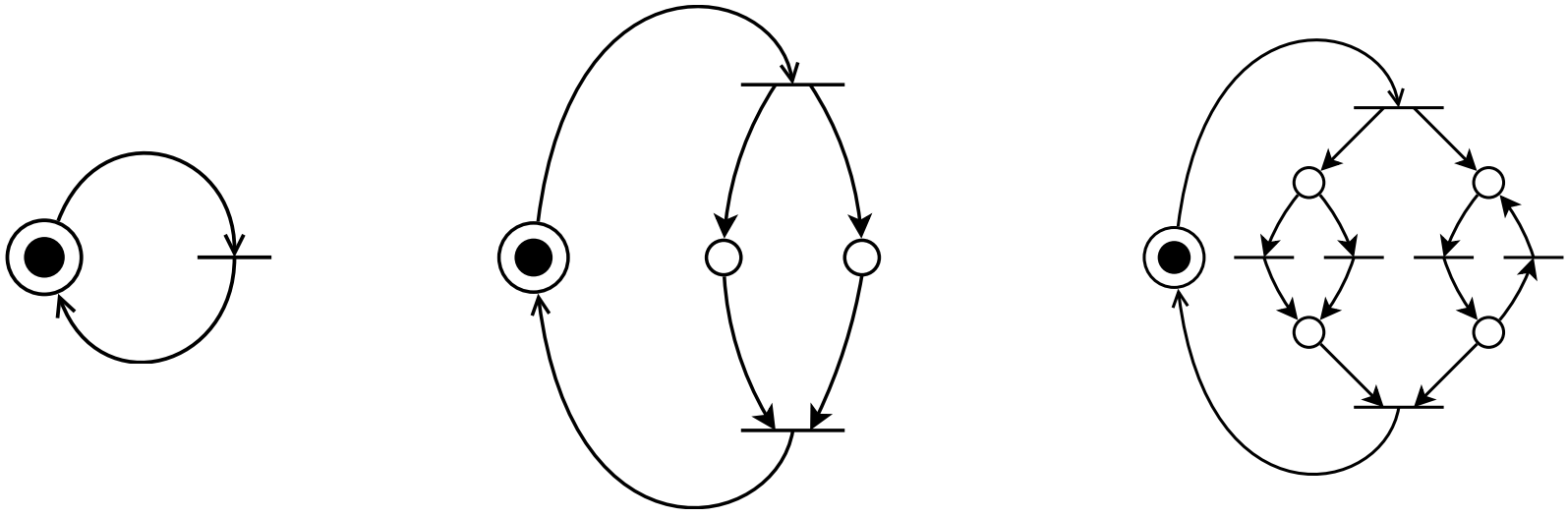
Type-2 construction



Type-2 construction



Type-2 construction



Outline

Compiler flow

Static Single Assignment (SSA)

```
d = m + n;
b = m - n;
if (b > 0) {
  a = b + c;
  d = e + a;
}
x = d + 2;
```

Sample program in C

```
A1 = m1 + n1;
S1 = m1 - n1;
C1: if (S1 > 0) {
  A2 = S1 + c1;
  A3 = e1 + A2;
}
M = φ(A1, A3);
A4 = M + 2;
```

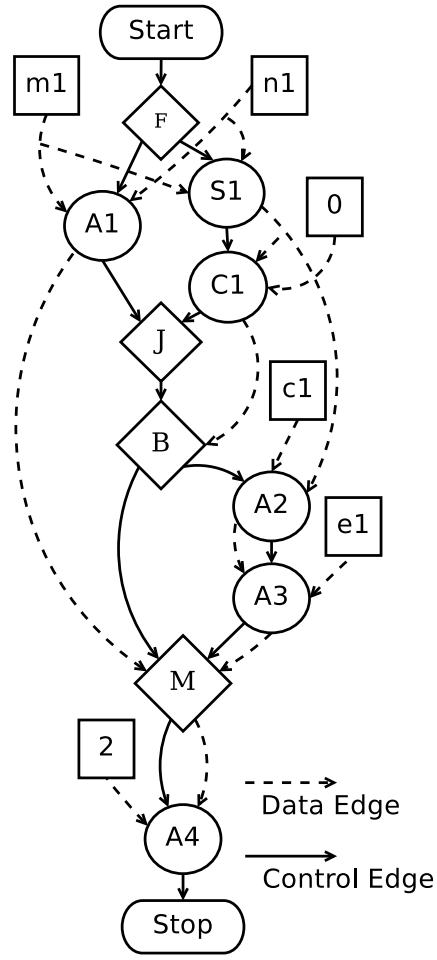
SSA form

Compiler flow

Control Data Flow Graphs (CDFG)

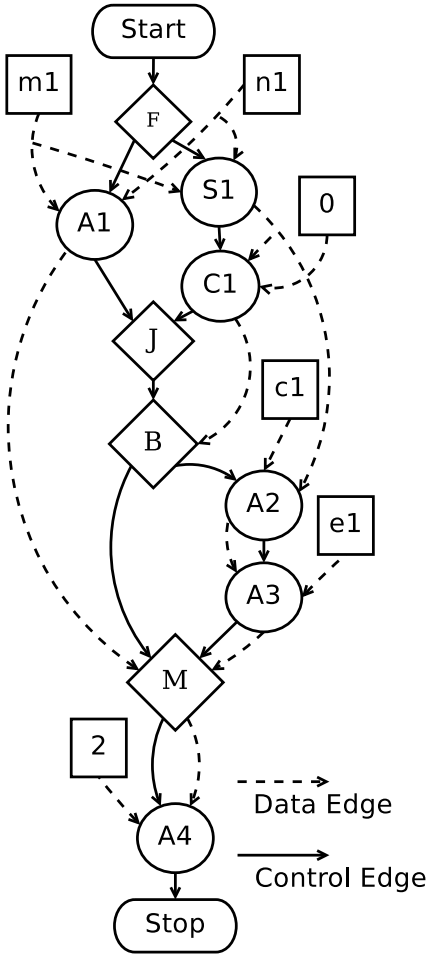
```
A1 = m1 + n1;
S1 = m1 - n1;
C1: if (S1 > 0) {
   A2 = S1 + c1;
   A3 = e1 + A2;
}
M = φ(A1, A3);
A4 = M + 2;
```

SSA form

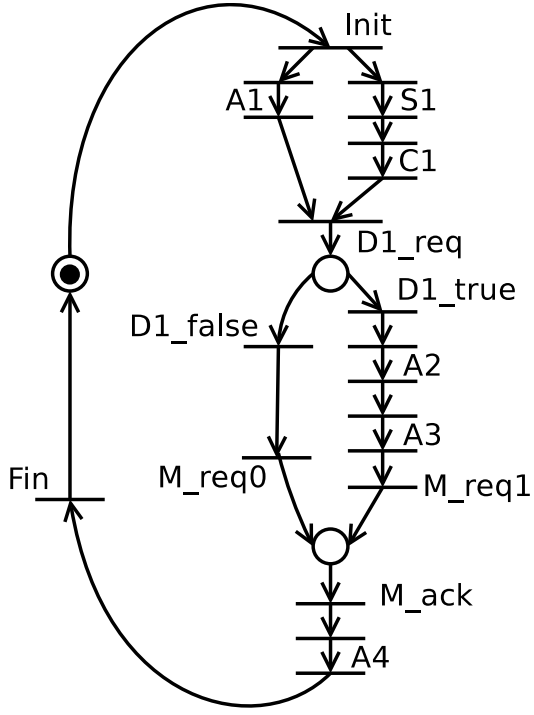


Control Data Flow Graph

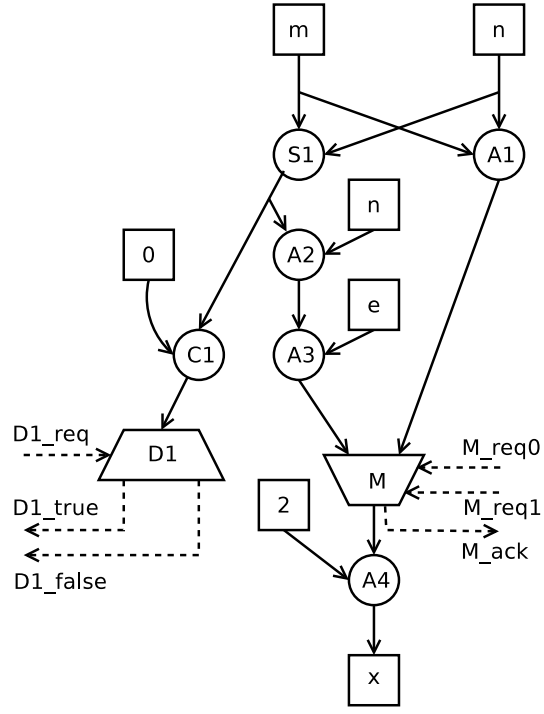
Control-path and data-path in AHIR



CDFG



Control Path

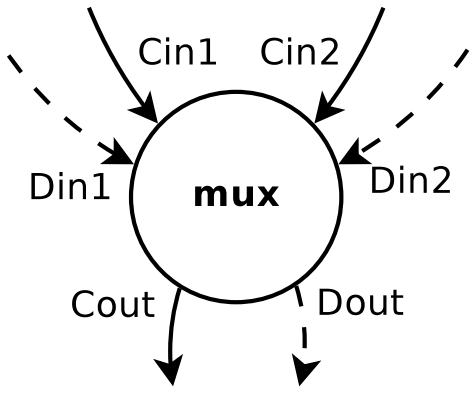


Data Path

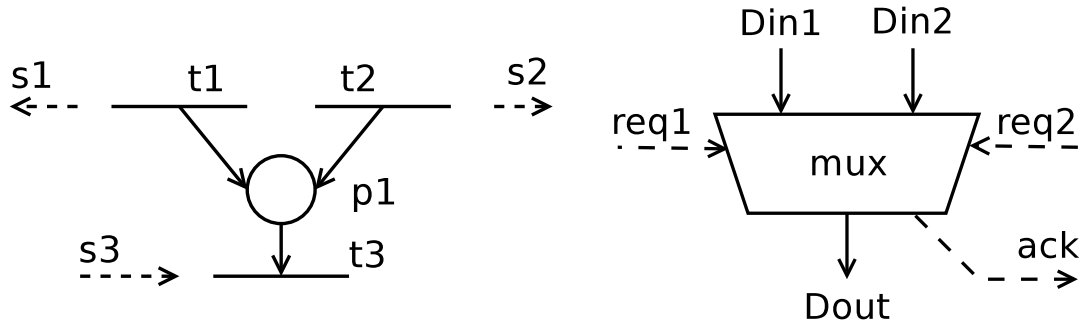
Construction

$$G \longrightarrow A$$

Each element in *G* is replaced with an *equivalent* AHIR fragment in *A*.



Multiplexer node



AHIR fragment

Construction

$$G \longrightarrow A$$

Each element in G is replaced with an *equivalent* AHIR fragment in A.

A is equivalent to G

For an initial state X

$S_G(X)$: Possible sequences of operations in G.

$S_A(X)$: Possible sequences of operations in A.

One-to-one correspondence between sets $S_G(X)$ and $S_A(X)$.

- No loss of information when translating a CDFG to AHIR.

Equivalence

Fragments in *A* correctly implement the elements in *G*.

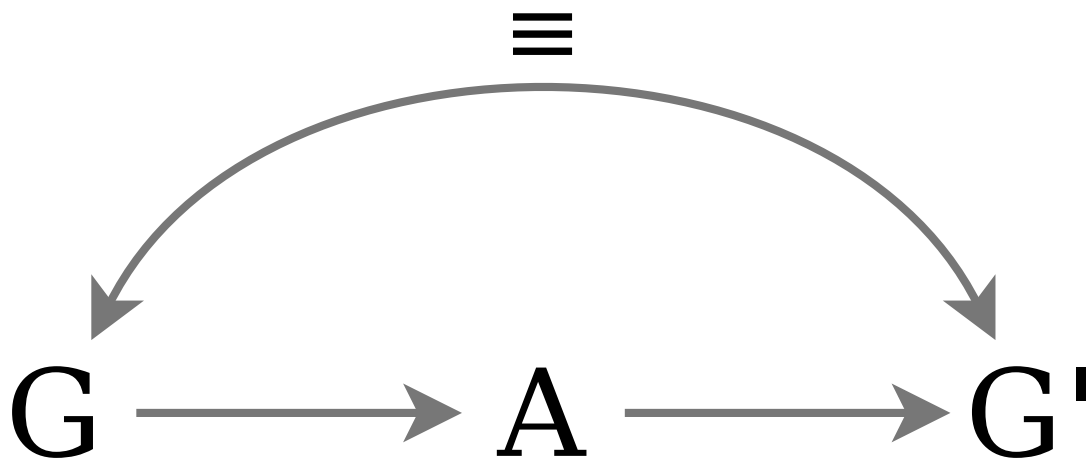
Sequences in *A* are included in the sequences in *G*.

Equivalence

Elements in G' correctly implement fragments in A.

Sequences in G' are included in the sequences in A.

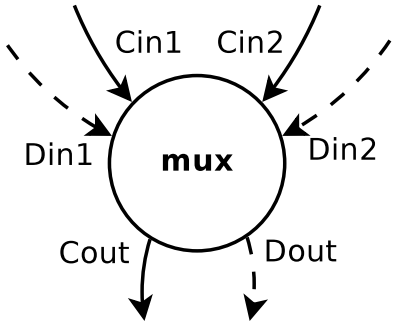
Equivalence



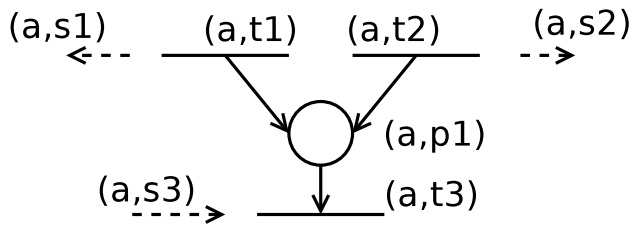
G′ is isomorphic to *G*.

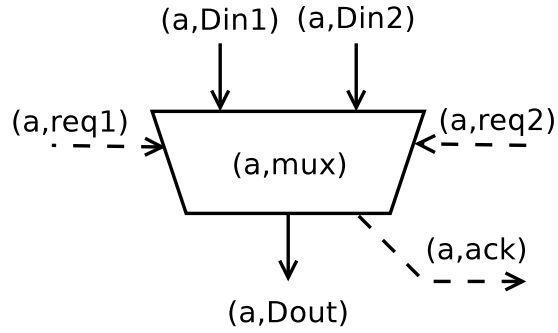
A is equivalent to *G*.

Reconstruction using labels



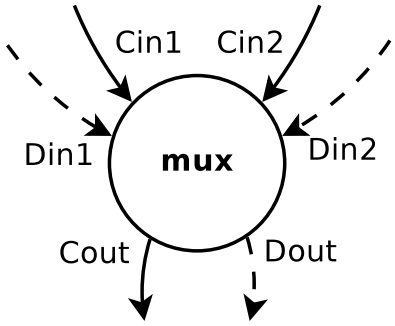
Multiplexer



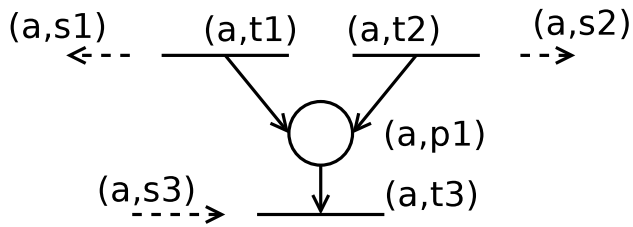


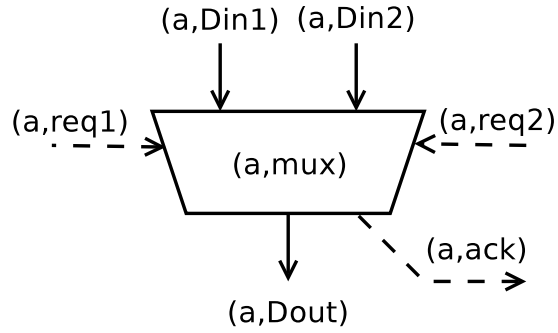
AHIR fragment

Reconstruction using labels



Multiplexer



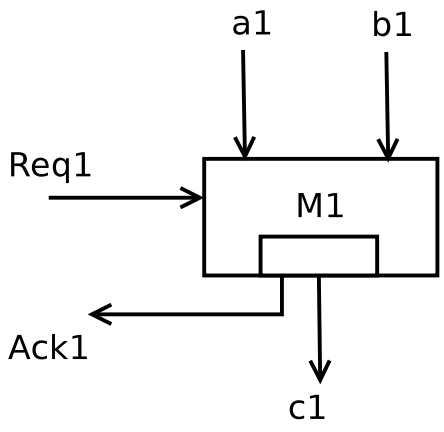


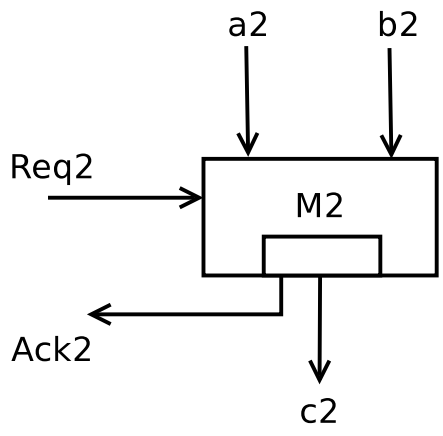
AHIR fragment

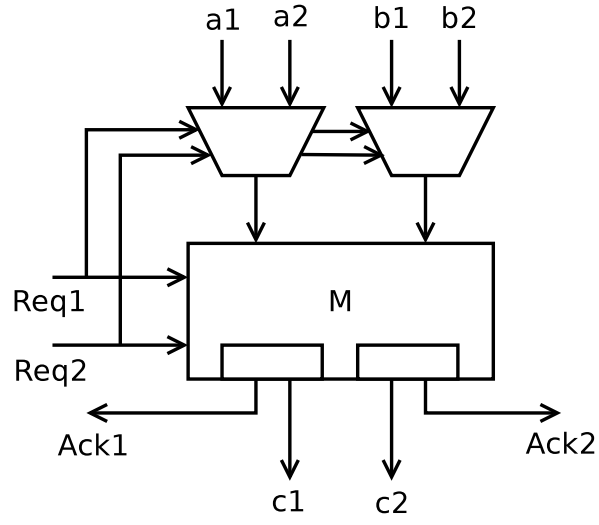
Labelled CDFG *G* → Labelled AHIR spec *A* → Labelled CDFG *G′*

Outline

Arbiterless sharing







Compatible operations

Two operations are compatible if they do not overlap in time.

Compatible operations

Two operations are compatible if they do not overlap in time.

- Compatibility in terms of paths in a Type-2 petri-net.

Compatible operations

Two operations are compatible if they do not overlap in time.

- Compatibility in terms of paths in a Type-2 petri-net.
- A labelling scheme to determine compatibility.

Compatible operations

Two operations are compatible if they do not overlap in time.

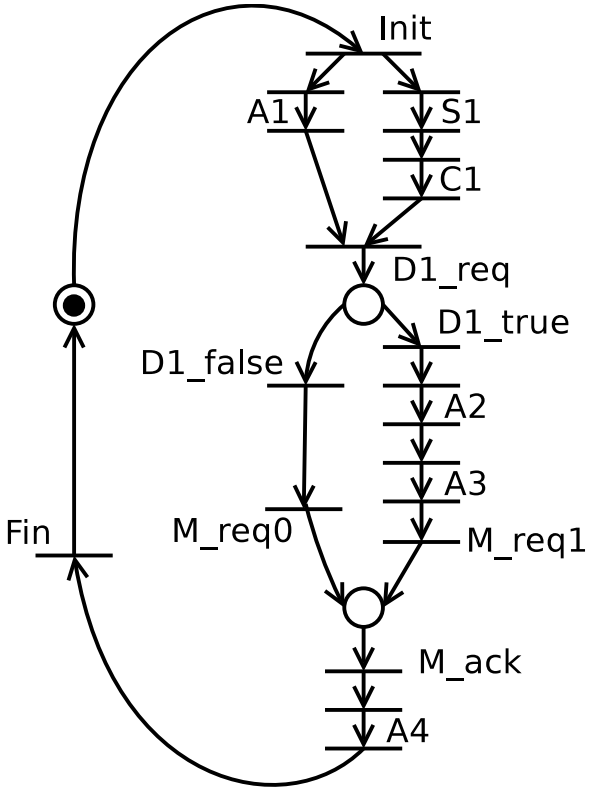
- Compatibility in terms of paths in a Type-2 petri-net.
- A labelling scheme to determine compatibility.
- An efficient graphical representation for labels.

Compatible operations

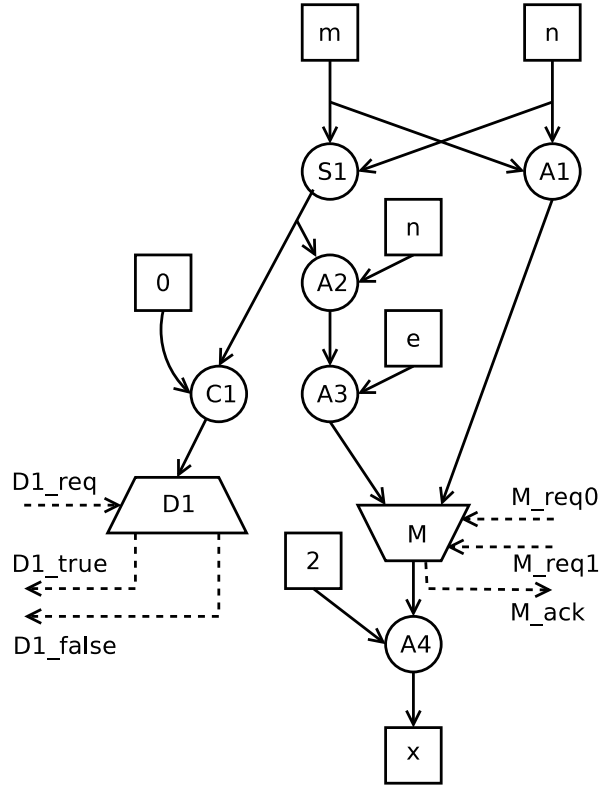
Two operations are compatible if they do not overlap in time.

- Compatibility in terms of paths in a Type-2 petri-net.
- A labelling scheme to determine compatibility.
- An efficient graphical representation for labels.
- A linear-time algorithm to check for compatible labels.

Compatible operations

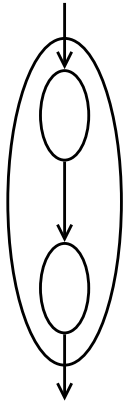


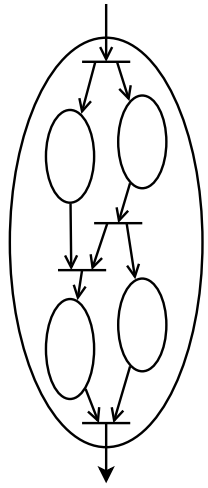
Control Path

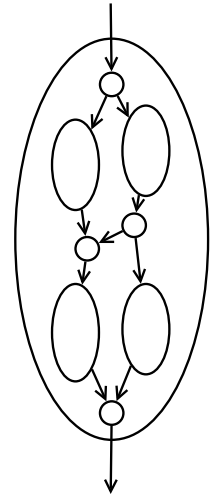


Data Path

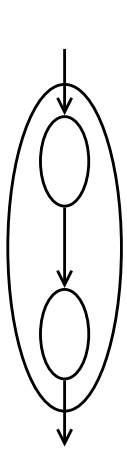
Compatibility in a Type-2 Petri-net

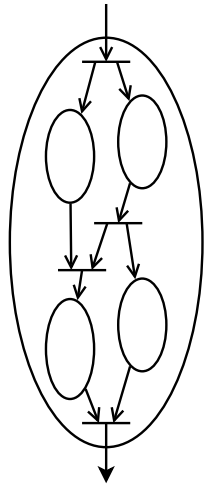


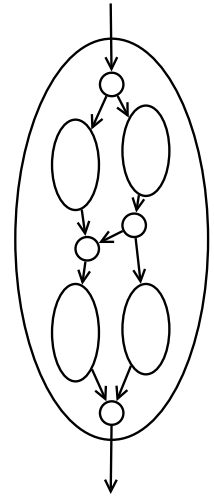




Compatibility in a Type-2 Petri-net





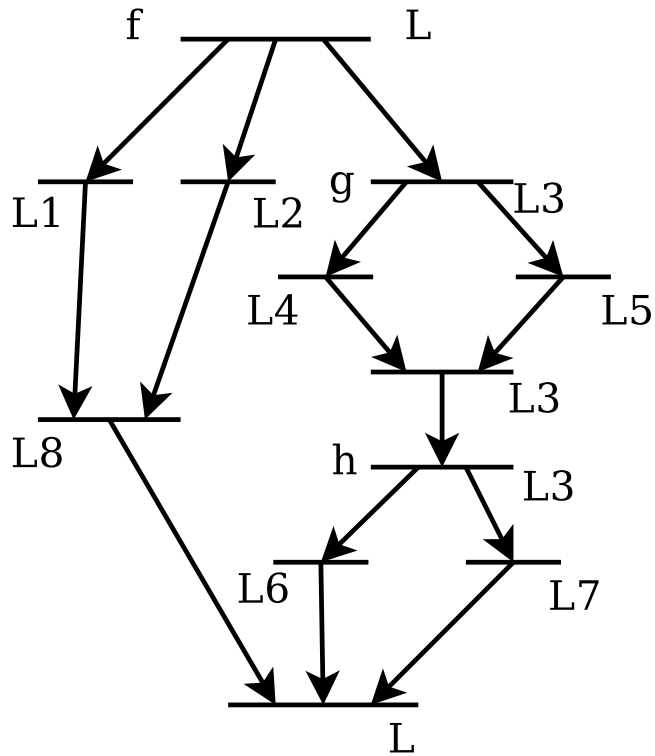


Two elements e_1 and e_2 in a Type-2 petri-net are compatible if and only if one of the following is true:

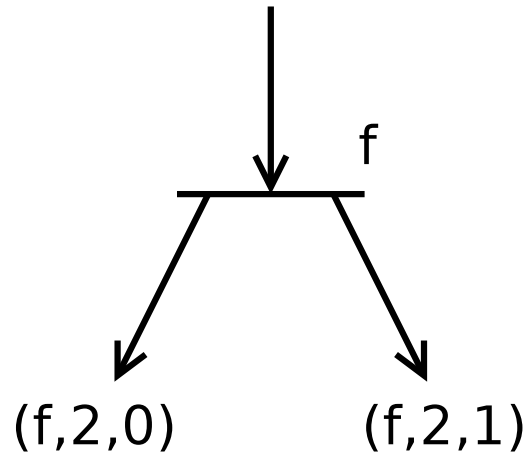
- The smallest region R that contains both of them is not a fork region.
- R is a fork region, but there is a path in R that passes through regions both e_1 and e_2.

Labelling scheme

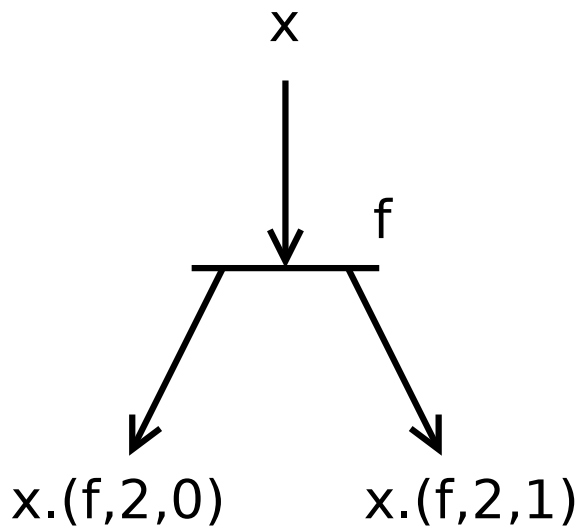
- Symbolic execution of the petri-net.
- Trace the influence of forks and joins.



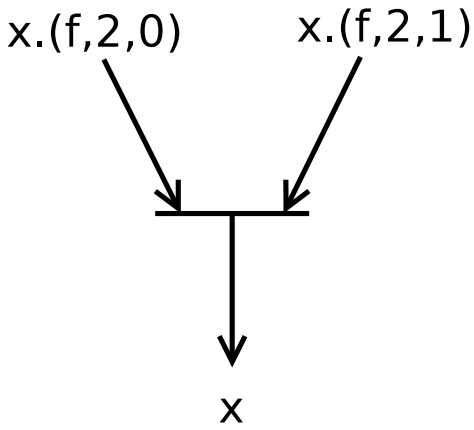
Labels at a fork

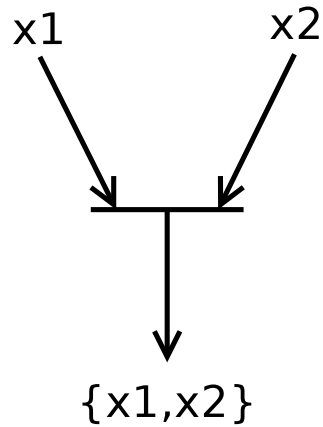


Labels at a fork



Labels at a join



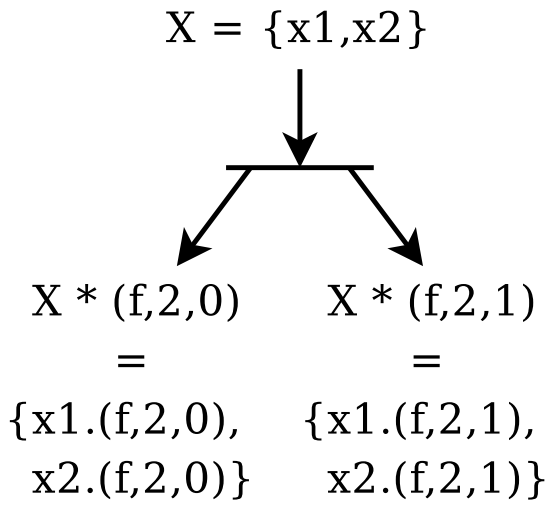


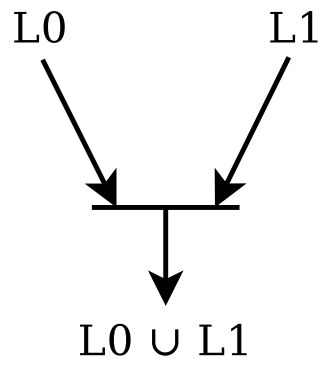
Labels

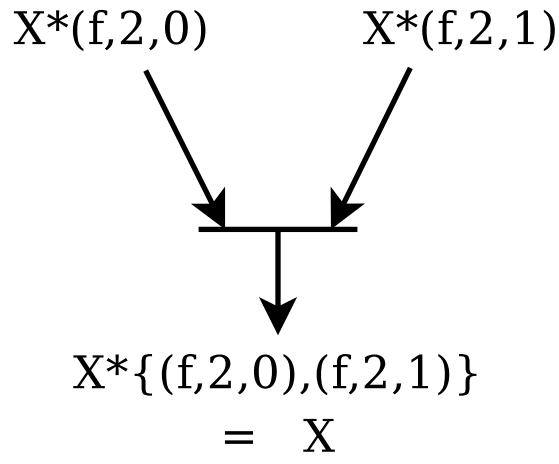
- A label is a set of sequences of label elements.
- A label element is a 3-tuple: $\left(f, k(f), i\right)$

$$
\begin{aligned}
f &: \text{fork identifier} \\
k(f) &: \text{fan-out of the fork } f \\
i &: \text{index into the fan-out of the fork}
\end{aligned}
$$

Label Operations







Compatibility of labels

Two operations in a Type-2 petri-net, with labels L_1 and L_2 respectively, are compatible if and only if the labels L_1 and L_2 are compatible.

Complexity

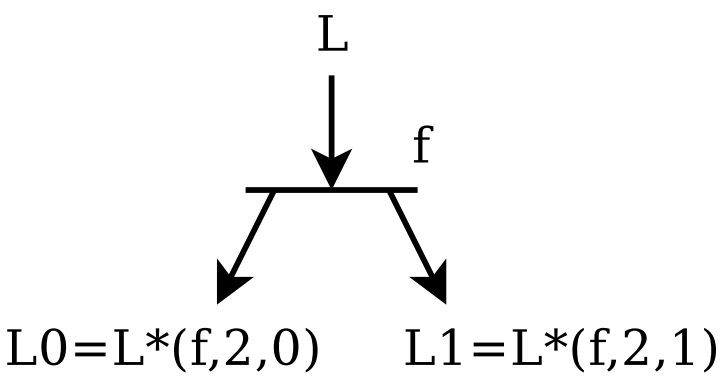
- The size of a compatibility label is exponential.
- Comparison of two labels is expensive.

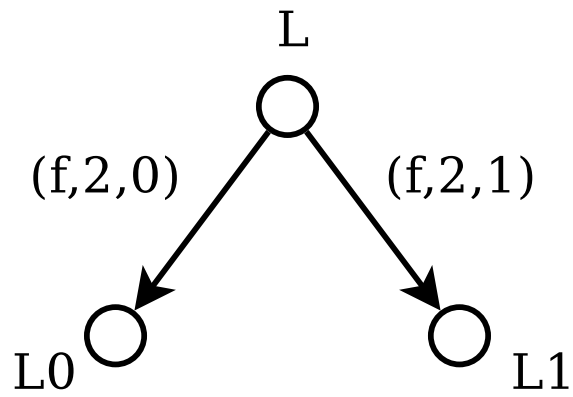
The Label Representation Graph (LRG)

- Nodes represent labels.
- Edges represent the construction of labels.
- The number of nodes is less than the number of elements in the Type-2 petri-nets.
- A linear-time algorithm to determine compatibility.

LRG Construction

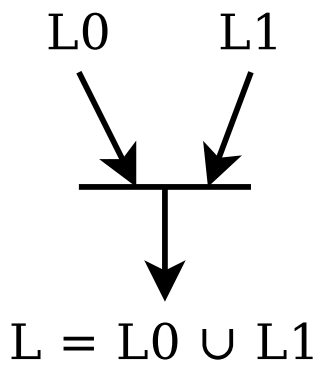
Labelled edges at a fork

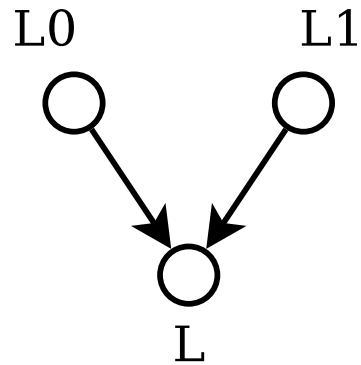




LRG Construction

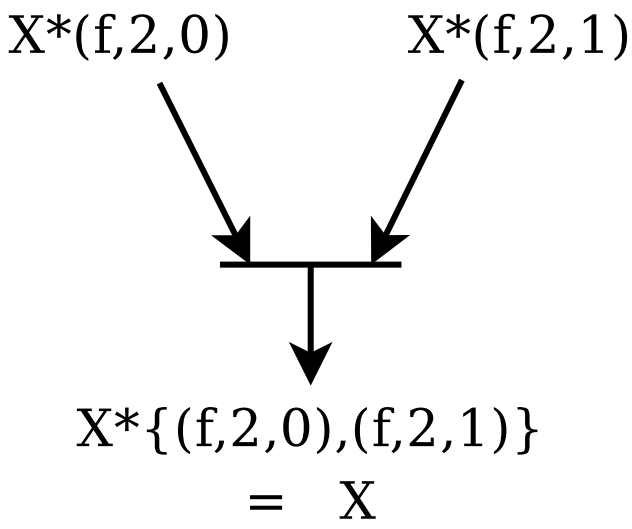
Unlabelled edges at a join

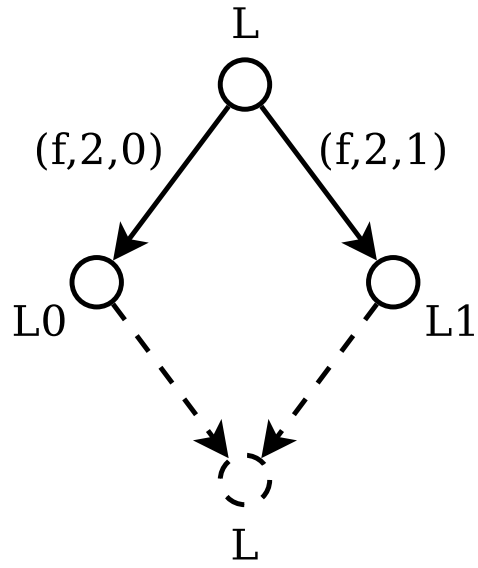




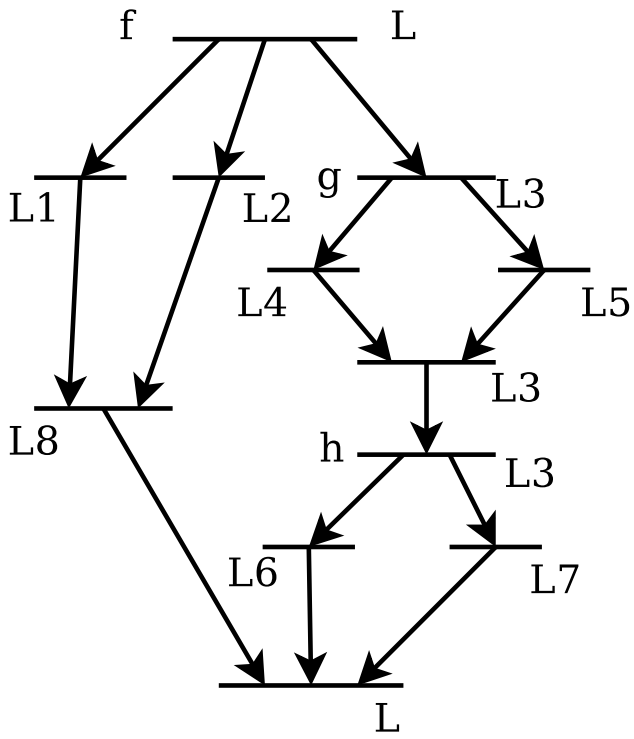
LRG Construction

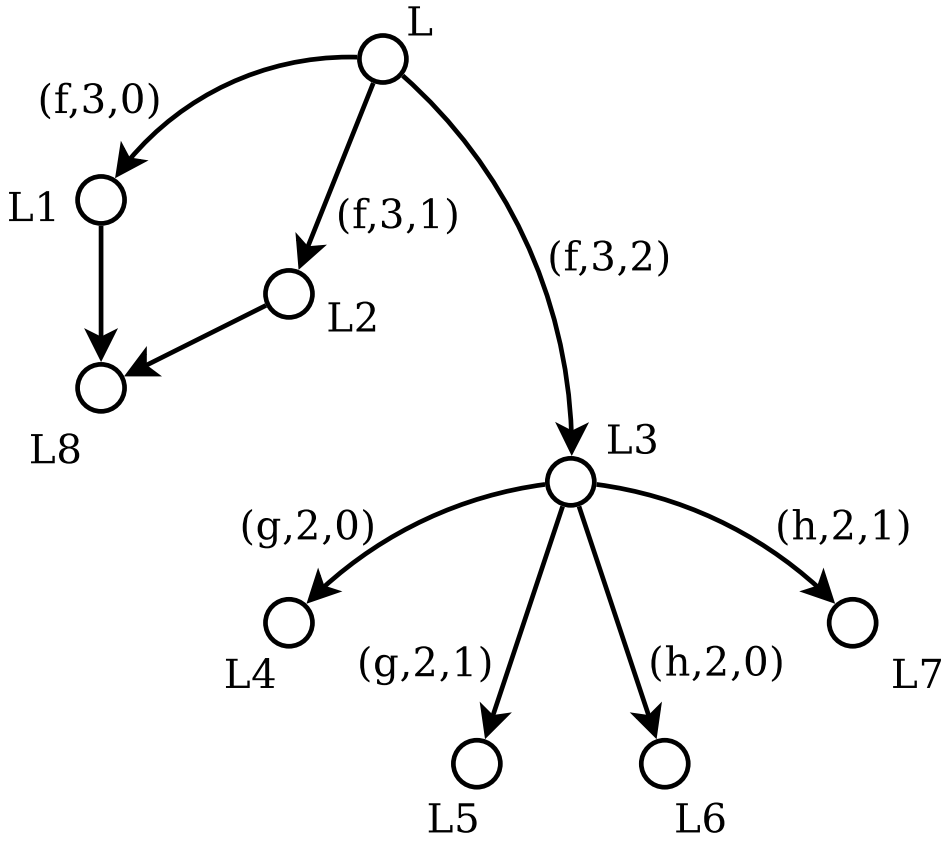
Reduction at a join





An example LRG





Compatibility using the LRG

Two labels L_1 and L_2 are compatible if and only if the corresponding nodes u and v in the LRG are compatible.

Compatibility using the LRG

Two labels L_1 and L_2 are compatible if and only if the corresponding nodes u and v in the LRG are compatible.

Complexity

- The number of nodes in the LRG is much less than the number of operations in the control-path.
- DFS-based algorithm to check a pair of nodes in the LRG.

Outline

Current Implementation

A complete flow from C to synthesisable VHDL using LLVM.

- 32-bit integers, single precision floats.
- Pointers, structures, arrays.
- Fixed number of arguments in function calls.
- Dynamic memory allocation.

Conceivable, but not implemented:

- Recursion.
- Function pointers.
- I/O ports.
- Variable number of arguments in function calls.
- Dynamically loaded libraries.

Synthesisable VHDL

- Clocked implementation using a predefined library.
- Asynchronous control-path and link-layer.
- Synchronous data-path operators
  - All operations take one clock cycle.
- Separate memory subsystem
  - Multiple load/store ports.
  - Unspecified number of clock cycles.

Examples used

- A5/1 stream cipher*
- AES block cipher*
- FFT
- Linpack
- Red-Black Trees

The same source code was run on a processor, and also synthesised to hardware.

*Based on work done by Prakalp Somawanshi, 2008

Arbiterless sharing

FPGA implementation (Xilinx Virtex II)

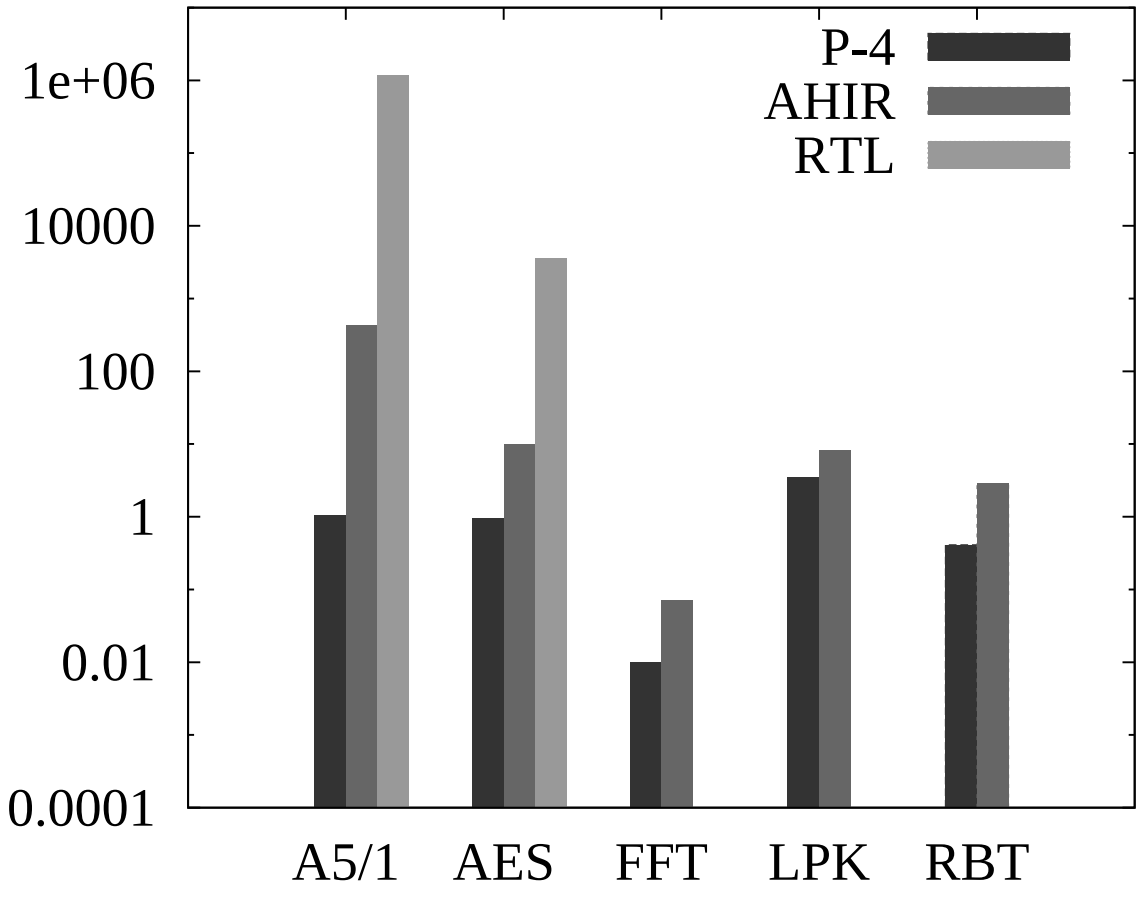
Performance / Area (Hz per slice)

Example	Gain (%)
A5/1	15
AES	126
FFT	97
RBT	191
LPK	36

Throughput / Area

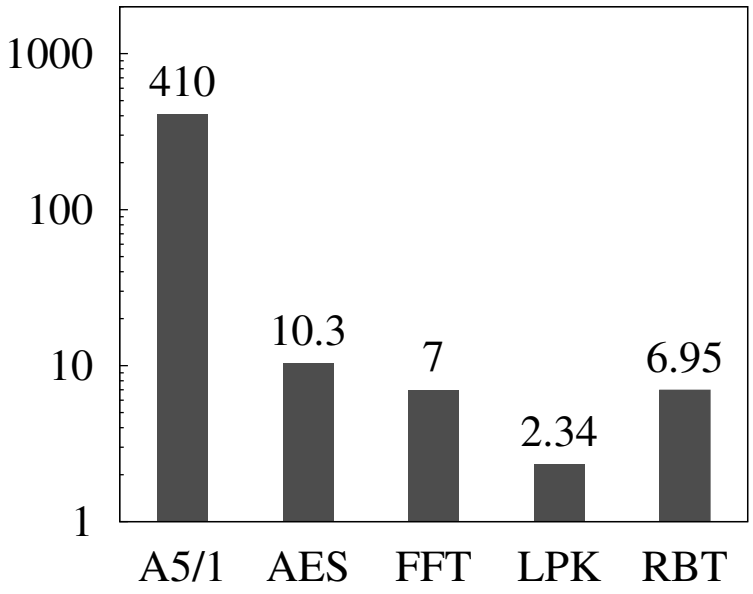
FPGA implementation (Xilinx Virtex II)

Comparison with Pentium IV and hand-crafted RTL

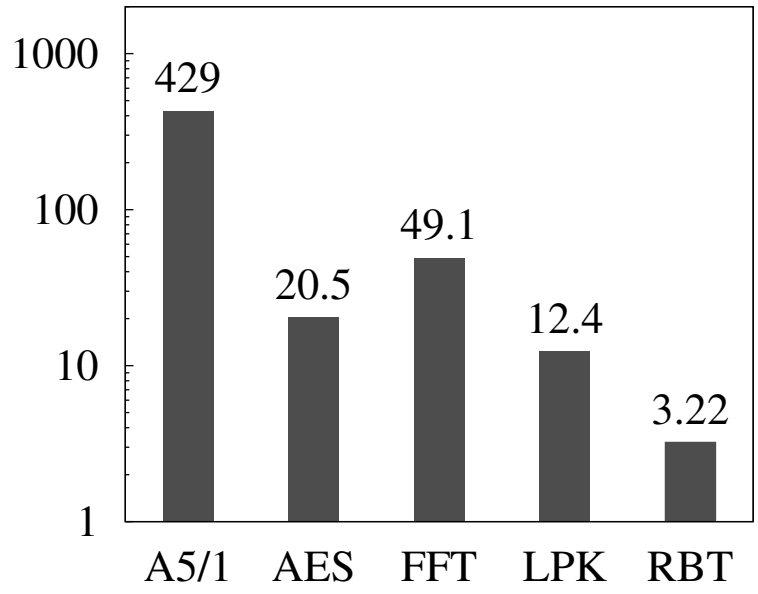


Throughput / Area

Comparison with processors



FPGA v/s Intel P-IV

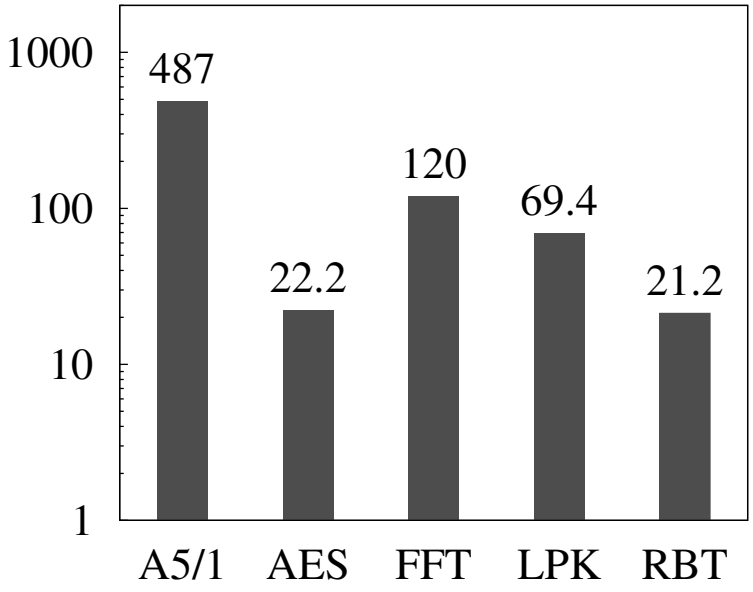


ASIC v/s Intel Atom N270*

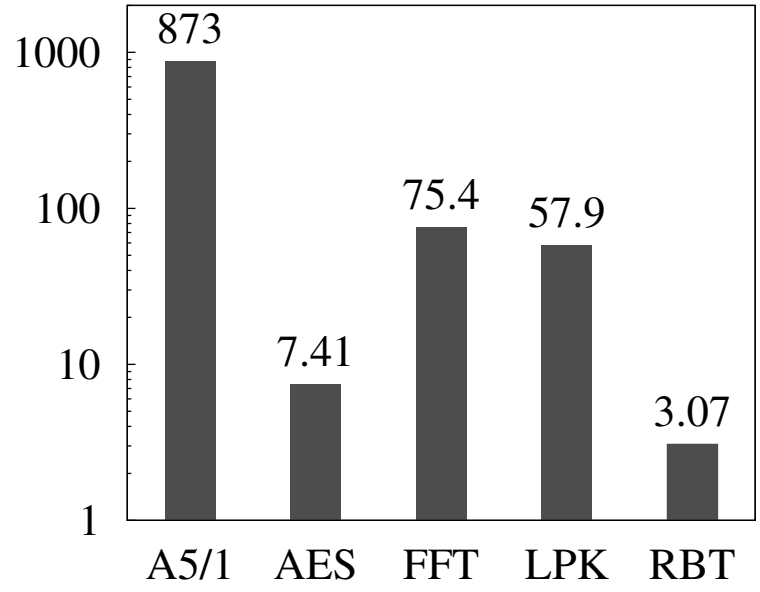
*Based on work done by Sreenivas S. and Kunal Ghosh, 2009

Energy

ASIC v/s Intel Atom N270



Energy
(performance per watt)



Energy $\times$ Delay

Conclusion

Established a high-level synthesis flow.

1. Supports a large class of programming languages.
2. Eliminates verification
   - Generated circuits are correct by construction.
3. Supports scalable optimisations.
   - Orthogonal factoring into control, data and storage.
   - Static analysis that exploits the structure of the petri-net.
4. Compares favourably with embedded processors.
   - Throughput / Area
   - Energy
   - Energy $\times$ Delay

Future work

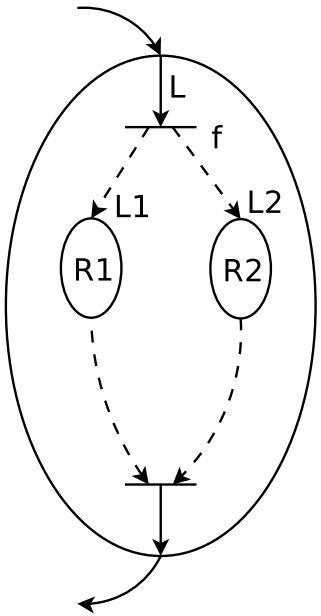
Getting close to hand-crafted hardware.

- Efficient hardware
  - Better operators
  - Hardware reuse
    - Routing overheads
    - Sharing registers
  - System-level parallelisation and pipe-lining
- Support for “better” languages
  - System-C
  - Esterel
  - Functional programming languages
- Customised memory subsystem
  - Relation between ports, banks and addresses
  - Memory access information available in the software phase.
  - Analyse traces for run-time access patterns

Thank You.

Compatibility and labels

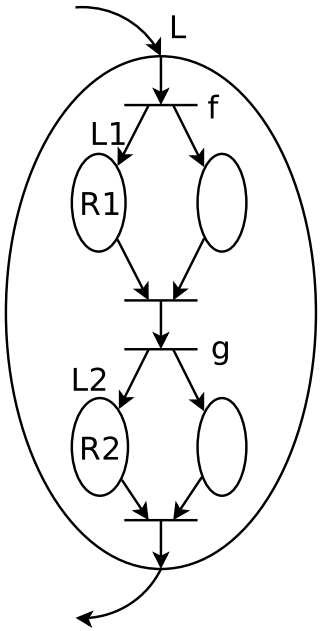
A simple fork



$$
\begin{aligned}
x &\in L \\
x.(f,2,0) &\in L1 \\
x.(f,2,1) &\in L2
\end{aligned}
$$

Compatibility and labels

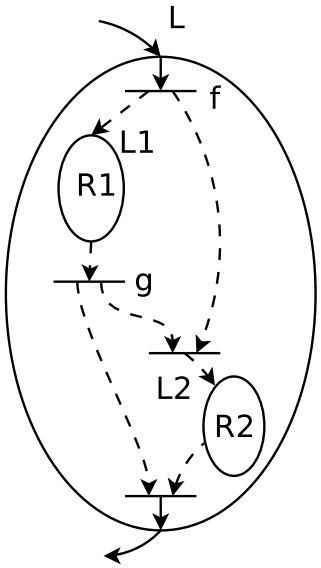
Two forks in series



$$
\begin{aligned}
x &\in L \\
x.(f,2,0) &\in L1 \\
x.(g,2,0) &\in L2
\end{aligned}
$$

Compatibility and labels

Overlapped forks



$$
\begin{aligned}
x &\in L \\
x.(f,2,0) &\in L1 \\
x.(f,2,1), &\in L2 \\
x.(f,2,0).(g,2,1) &
\end{aligned}
$$

Compatible Labels

Definition: Two labels L_1 and L_2 are compatible if and only if there exist sequences $l_1 \in L_1$ and $l_2 \in L_2$ where one of the following is true:

1. l_1 and l_2 are equal.
2. One of them is a prefix of the other.
   - $x.(f, 2, 0)$ and $x.(f, 2, 0).(g, 2, 1)$
3. The elements that occur in them just after the longest common prefix disagree on the forks.
   - $x.(f, 2, 0).(h, 2, 1)$ and $x.(f, 2, 0).(g, 2, 1)$

Examples

Compatible pairs:

1. $(f, 2, 0)$, $(f, 2, 0)$
2. $(f, 2, 0)$, $(f, 2, 0).(g, 2, 1)$
3. $(f, 2, 0).(h, 2, 1)$, $(f, 2, 0).(g, 2, 1)$

Incompatible pairs:

1. $(f, 2, 0)$, $(f, 2, 1)$
2. $(f, 2, 0)$, $(f, 2, 1).(g, 2, 1)$
3. $(f, 2, 0).(g, 2, 0)$, $(f, 2, 0).(g, 2, 1)$

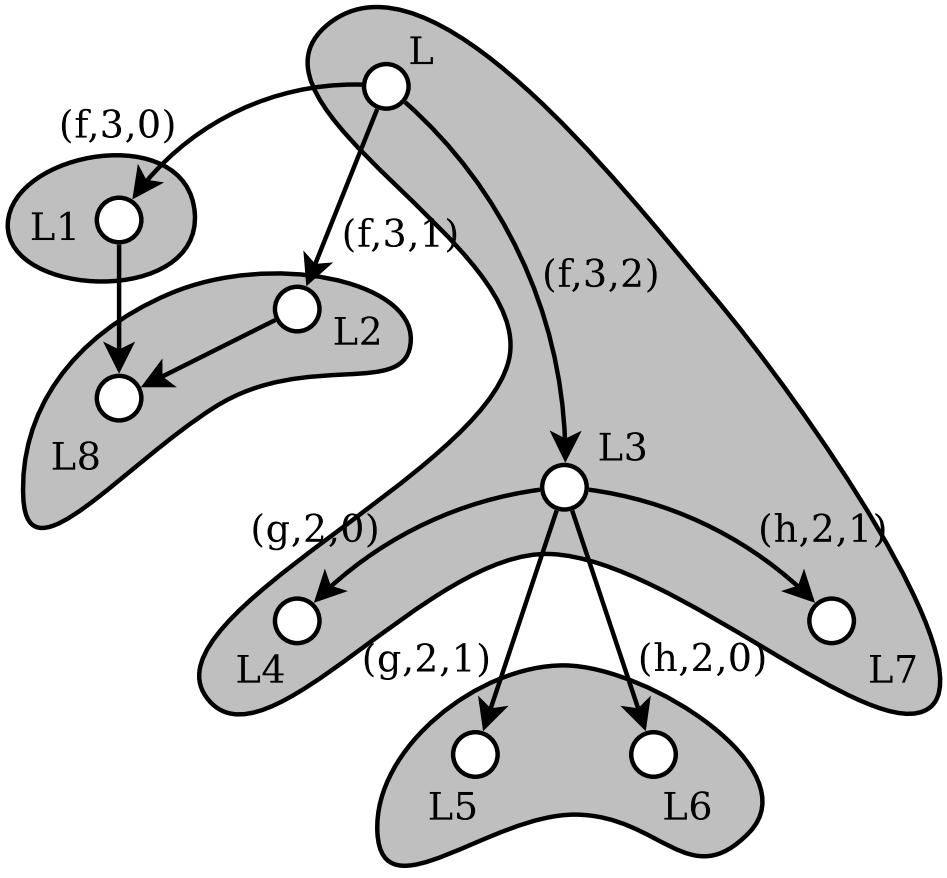
BACK

Compatibility using the LRG

Definition: Two nodes u and v in the Label Representation Graph are compatible if and only if one of the following is true:

1. There is a path between u and v.
2. They are reachable from a common ancestor a in the LRG along paths such that one of the following is true:
   - 2.1 One or both paths begin with an unlabelled edge.
   - 2.2 The labels on the first edges in the paths indicate different forks.

Compatibility cover



BACK

AHIR v/s Atom

	Area (mm^2)	Freq (MHz)	Delay (ms)	Power (mW)	Energy (μJ)	E$\times$D
A5/1-Atom	25	1600	0.12μs	2500	298.44 nJ	35.63
A5/1-AHIR	0.10	285	0.07μs	9.22	0.61 nJ	0.04
AES-Atom	25	1600	0.036	2500	89.362	3.194
AES-AHIR	0.41	285	0.107	37.56	4.023	0.431
FFT-Atom	25	1600	0.022	2500	55.64	1.238
FFT-AHIR	0.32	180	0.035	13.11	0.464	0.016
LPK-Atom	25	1600	7.90	2500	19740	155875
LPK-AHIR	1.69	165	9.42	30.33	285	2691
RBT-Atom	25	1600	0.36	2500	891.89	318.19
RBT-AHIR	1.13	165	2.47	17.00	42.01	103.80