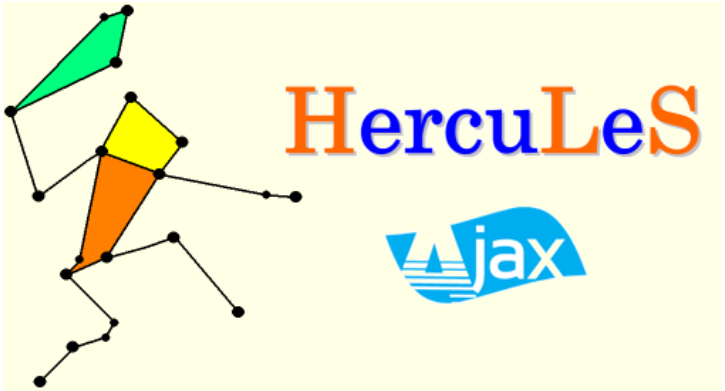
# Hardware design space exploration using **HercuLeS** HLS

Nikolaos Kavvadias
nkavvadias@ajaxcompilers.com

CEO, Ajax Compilers, Athens, Greece
www.ajaxcompilers.com



# The need for high-level synthesis (HLS)

- Moore's law anticipates an annual increase in chip complexity by 58%
- At the same time, human designer's productivity increase is limited to 21% per annum
- This designer-productivity gap is a major problem in achieving time-to-market of hardware products

Solution Adoption of a high-level design and synthesis methodology imposing user entry from a raised level of abstraction

- Hide low-level, time-consuming, error-prone details
- Drastically reduce human effort
- End-to-end automation from concept to production

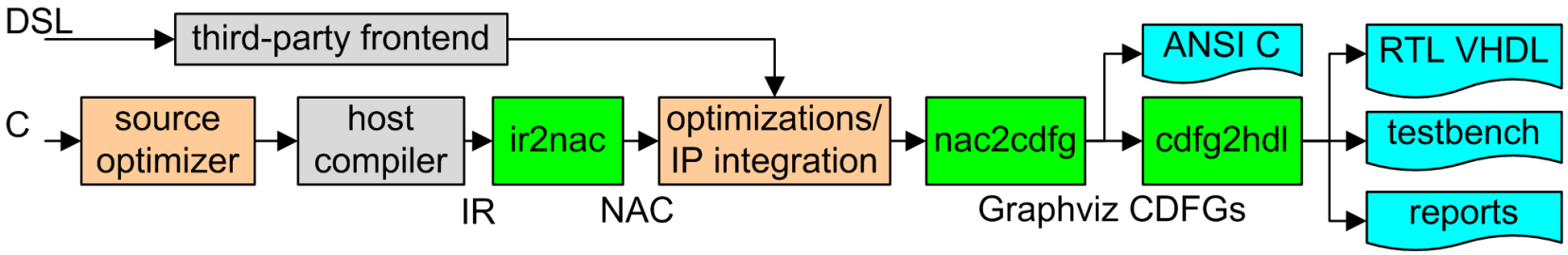
HLS An algorithmic description is automatically synthesized to a customized digital embedded system

# The HercuLeS environment

- HercuLeS is an easy to use, extensible, high-level synthesis environment for whole-program hardware compilation
- In development since 2009
- Marketed by Ajax Compilers
- HercuLeS targets both hardware and software engineers/developers
  1. ASIC/SoC developers, FPGA-based/prototype/reference system engineers
  2. Algorithm developers (custom hardware algorithm implementations)
  3. Application engineers (application acceleration)
- Terminology

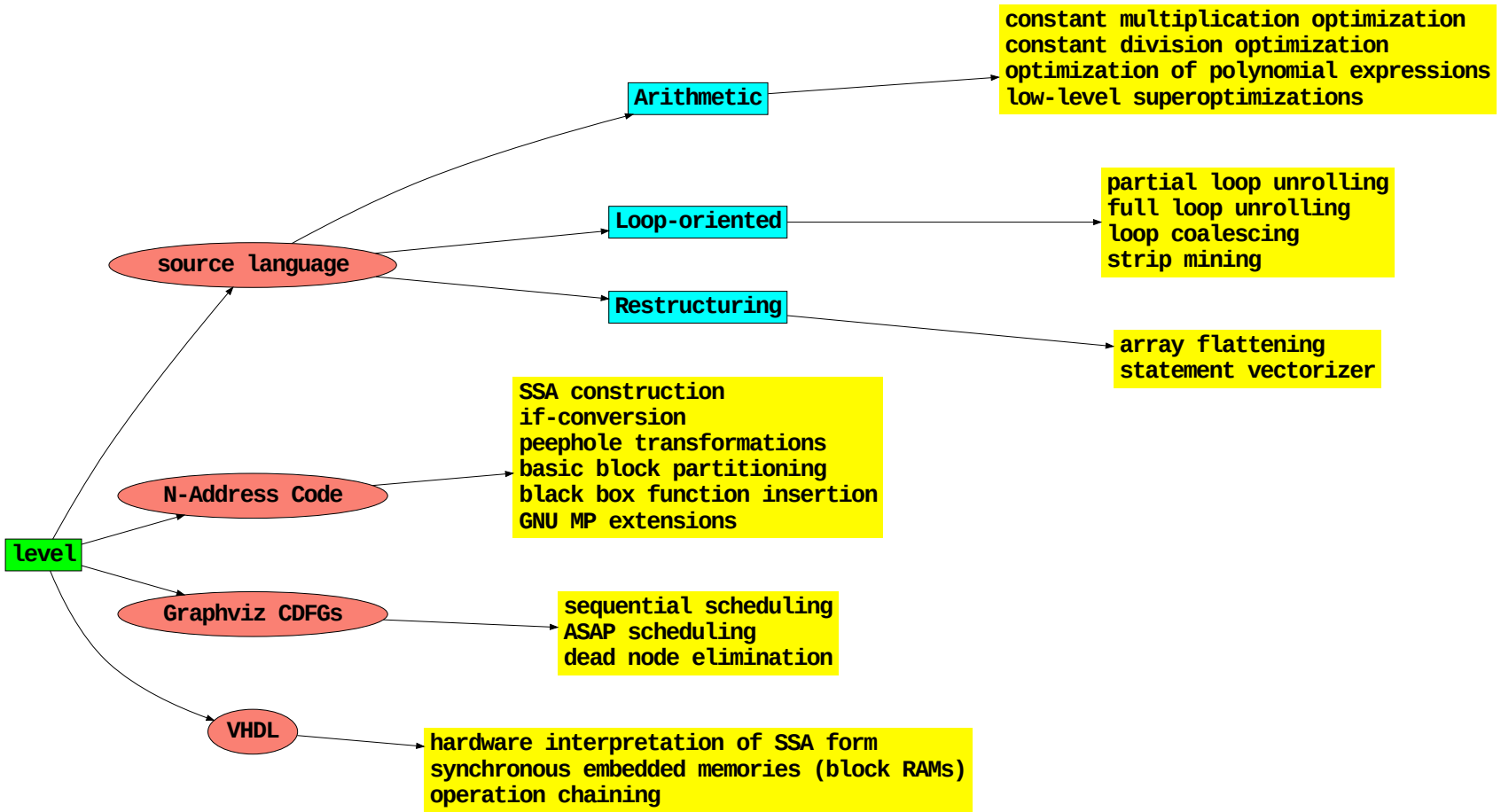
| | |
|---|---|
| ASIC | Application-Specific Integrated Circuit |
| SoC | System-on-a-Chip |
| FPGA | Field-Programmable Gate Array (post-fabrication reconfigurable chip) |

# The HercuLeS flow



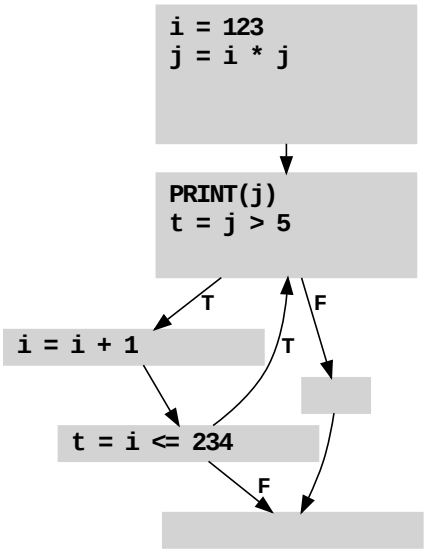

- Optimized C code is passed to the host compiler for compiler intermediate representation (IR) generation
- *ir2nac* translates to N-Address Code (NAC)
- IP components are automatically inserted using black box function calls
- HercuLeS = *nac2cdfg* + *cdfg2hdl*
  - *nac2cdfg*: CDFG (Control-Data Flow Graph) extraction
  - *cdfg2hdl*: hardware and self-checking testbench generation
- ANSI C backend for rapid algorithm verification
- Optimizations at the source, NAC, Graphviz, and VHDL levels

# The optimization space of HercuLeS

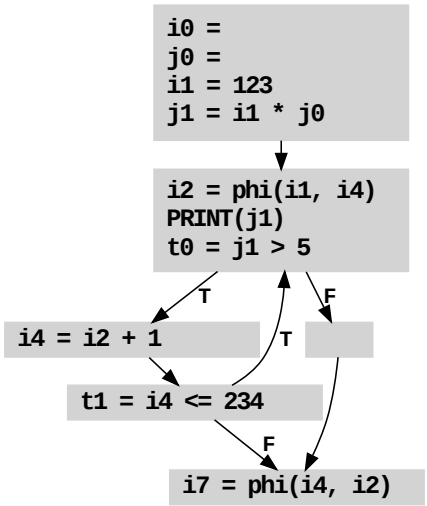


# SSA (Single Static Assignment) form construction

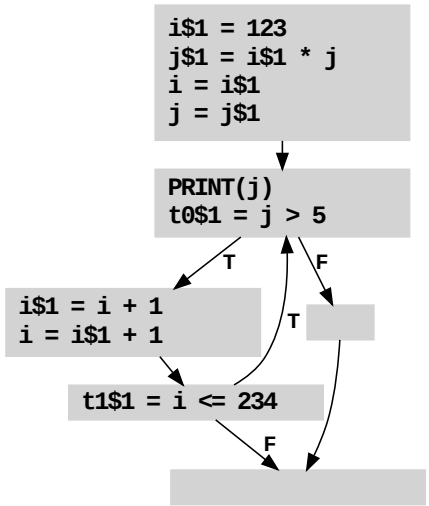
- Pseudo-statements called $\phi$-functions join variable definitions from different control-flow paths
- Enforce a single definition site for each variable
- False dependencies are naturally removed
- Many analyses and optimizations are simplified
- HercuLeS supports minimal SSA (in the number of $\phi$s) and intrablock-only (pseudo) SSA




Prior SSA




Minimal SSA




Pseudo SSA

# Representing hardware as FSMDs

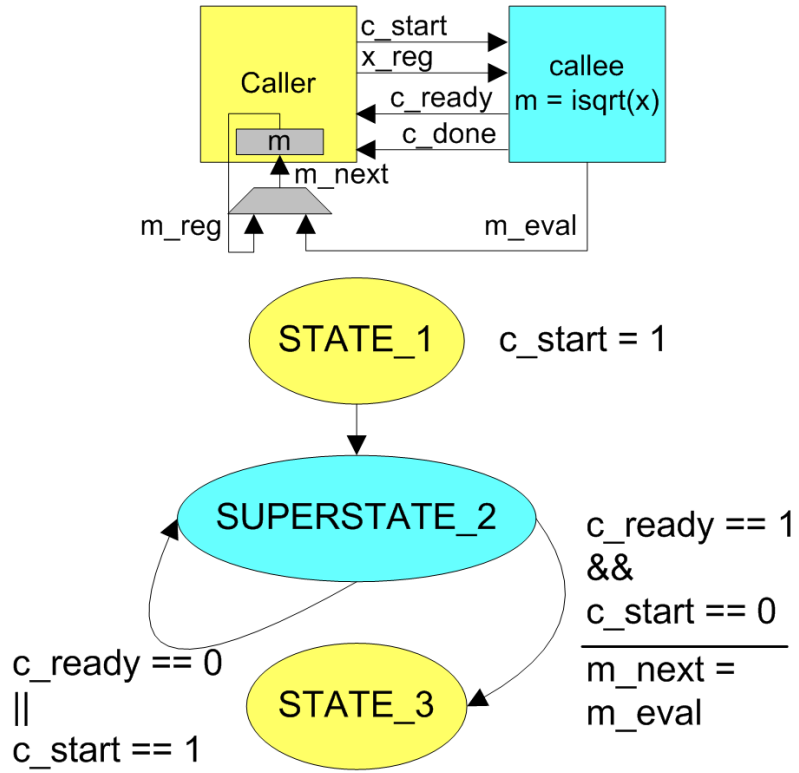

I/O interface

| Port | Dir. | Description |
|---|---|---|
| clk | I | external clocking source |
| reset | I | asynchronous (or synchronous) reset |
| start | I | enable computation |
| din | I | data inputs |
| dout | O | data outputs |
| ready | O | the block is ready to accept new input |
| valid | O | a data output port is streamed out |
| done | O | end of computation for the block |

- FSMD (Finite-State Machine with Datapath) as a MoC is universal, well-defined and suitable for either data- or control-dominated applications
- FSMDs = FSMs with embedded datapath actions
- HercuLeS supports extended FSMDs (hierarchical calls, communication with on-chip memories, IP integration)

# Hierarchical calls between FSMDs

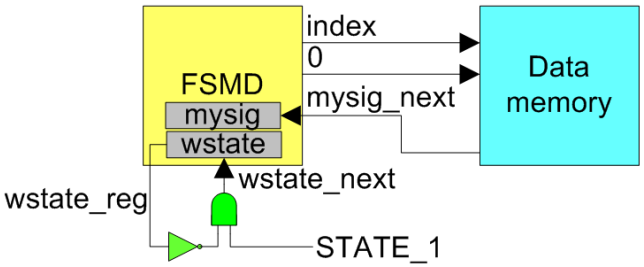
- Supported to arbitrary depth and complexity (apart from recursion)
- Example of a caller FSMD, handing over computation to callee superstate (a square root computation)
- Variable-related quantities are represented by three signals: `*_next` (value to-be-written), `*_reg` (value currently read from register), `*_eval` (callee output)

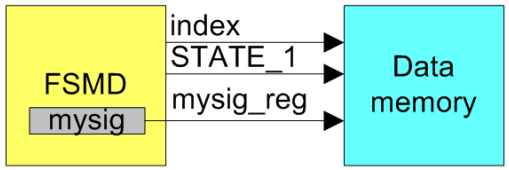


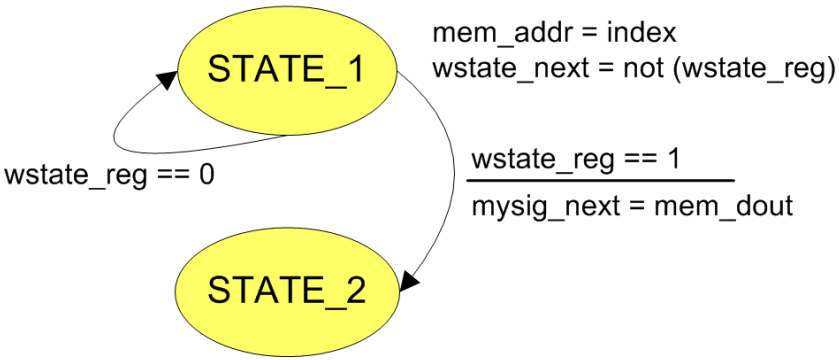
# Communication with embedded memories

**load** Requires a wait-state register for devising a dual-cycle substate (address + data cycles)

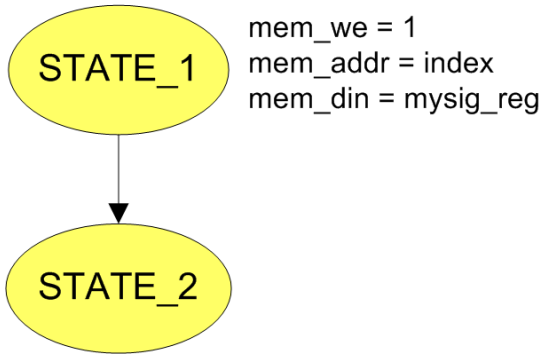
**store** Raises block RAM write. Stored data are made available in the subsequent machine cycle



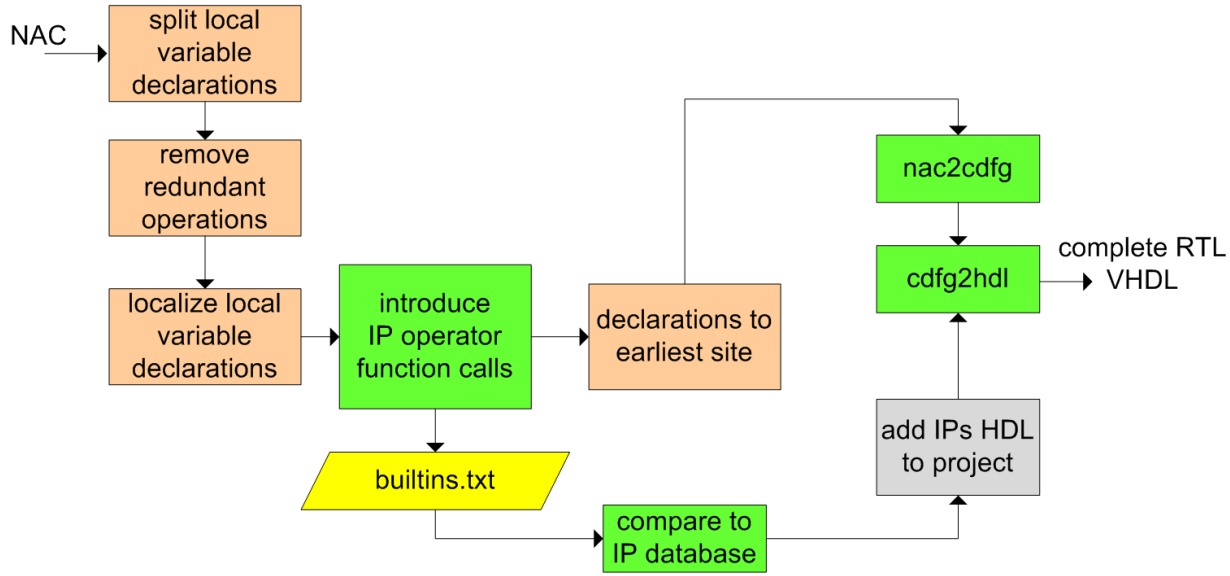




load




store

# Automatic IP integration




**IPs** Third-party components used in hardware systems (e.g. dividers, floating-point operators)

- How to import and use your own IP
  1. Implement IP and place in proper subdirectory
  2. Add entry in text database
  3. Replace operator uses by black-box function calls
  4. HercuLeS creates a hierarchical FSMD with the requested callee(s)

# Example: Prime factorization (1/4)

- A streaming-output implementation of prime factorization

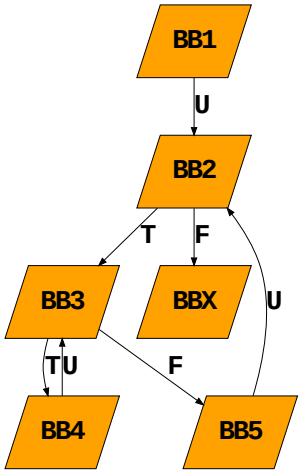
```
void pfactor(unsigned int x,
  unsigned int *outp)
{
  unsigned int i=2, n=x;
  while (i <= n) {
    while ((n % i) == 0) {
      n = n / i;
      *outp = i;
    }
    i = i + 1;
  }
}
```

ANSI C

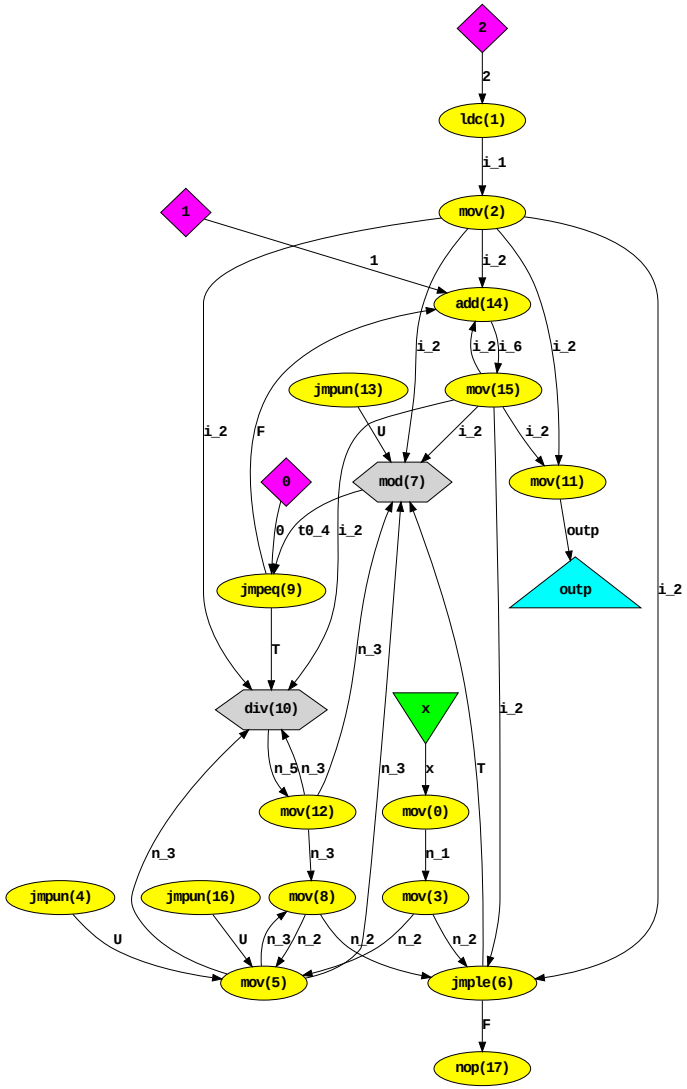
```
procedure pfactor(in u32 x, out u32 outp) {
  localvar u32 D_1369, i, n;
L0005:
  n <= mov x;
  i <= ldc 2;
  D_1366 <= jmpun;
D_1365:
  D_1363 <= jmpun;
D_1362:
  (n) <= divu(n, i);
  outp <= mov i;
  D_1363 <= jmpun;
D_1363:
  (D_1369) <= modu(n, i);
  D_1362, D_1364 <= jmpeq D_1369, 0;
D_1364:
  i <= add i, 1;
  D_1366 <= jmpun;
D_1366:
  D_1365, D_1367 <= jmple i, n;
D_1367:
  nop;
}
```

NAC

# Example: Prime factorization (2/4)




CFG




CDFG

# Example: Prime factorization (3/4)

```
when S_002_001 =>
  modu_10_start <= '1';
  next_state <= S_004_001;
when S_003_001 =>
  if (divu_6_ready='1' and
      divu_6_start='0') then
    n_1_next <= n_1_eval;
    next_state <= S_003_002;
  else
    next_state <= S_003_001;
  end if;
...
when S_004_001 =>
  if (modu_10_ready='1' and
      modu_10_start='0') then
    D_1369_1_next <= D_1369_1_eval;
    next_state <= S_004_002;
  else
    next_state <= S_004_001;
  end if;
when S_004_002 =>
  if (D_1369_1_reg = CNST_0) then
    divu_6_start <= '1';
    next_state <= S_003_001;
  else
    next_state <= S_005_001;
  end if;
```

VHDL

```
...
divu_6 : entity WORK.divu(fsmd)
  generic map (W => 32)
  port map (
    clk,
    reset,
    divu_6_start,
    n_reg,
    i_reg,
    n_1_eval,
    divu_6_done,
    divu_6_ready);

modu_10 : entity WORK.modu(fsmd)
  generic map (W => 32)
  port map (
    clk,
    reset,
    modu_10_start,
    n_reg,
    i_reg,
    D_1369_1_eval,
    modu_10_done,
    modu_10_ready);
```

VHDL (cont.)

# Example: Prime factorization (4/4)

```
00000004 00000002 00000002
00000005 00000005
00000006 00000002 00000003
00000007 00000007
00000008 00000002 00000002 00000002
00000009 00000003 00000003
0000000a 00000002 00000005
```

Input vector data

```
 x=00000004 outp=00000002 outp_ref=00000002
 x=00000004 outp=00000002 outp_ref=00000002
PFACTOR OK: Number of cycles=212
 x=00000005 outp=00000005 outp_ref=00000005
PFACTOR OK: Number of cycles=265
 x=00000006 outp=00000002 outp_ref=00000002
 x=00000006 outp=00000003 outp_ref=00000003
PFACTOR OK: Number of cycles=256
 x=00000007 outp=00000007 outp_ref=00000007
PFACTOR OK: Number of cycles=353
 x=00000008 outp=00000002 outp_ref=00000002
 x=00000008 outp=00000002 outp_ref=00000002
 x=00000008 outp=00000002 outp_ref=00000002
PFACTOR OK: Number of cycles=291
 x=00000009 outp=00000003 outp_ref=00000003
 x=00000009 outp=00000003 outp_ref=00000003
PFACTOR OK: Number of cycles=256
 x=0000000A outp=00000002 outp_ref=00000002
 x=0000000A outp=00000005 outp_ref=00000005
```

Diagnostic output

# Design space exploration configurations

- Explore different choices both in frontend translation (e.g. SSA construction) and hardware optimization
- Numerous configurations are possible
- The following configuration sets will be used
  - **O1** sequential scheduling
  - **O2** ASAP scheduling using SSA
  - **O3** **O2** with operation chaining (collapsing dependent operations to a single state)
  - **O4** **O3** with pseudo-SSA construction
  - **O5** **O3** with preserving $\phi$ functions
  - **O6** **O3** with block RAM inference

# Fibonacci series example: Introduction

- Fibonacci series computation is defined as

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

- Three iterative variants
  - **A** Addition and subtraction in the main loop
  - **B** Addition with one more temporary
  - **C** Addition with an in-situ register swap

```
uint32 fibo(uint32 x) {
  uint32 f0=0, f1=1, k=2;
#ifdef B
  uint32 f;
#endif
  do {
    k = k + 1;
#if A
    f1 = f1 + f0;
    f0 = f1 - f0;
#elif B
    f = f1 + f0;
    f0 = f1;
    f1 = f;
#elif C
    f0 = f1 + f0;
    SWAP(f0, f1);
#endif
  } while (k <= x);
#if A || B
  return (f1);
#else
  return (f0);
#endif
}
```

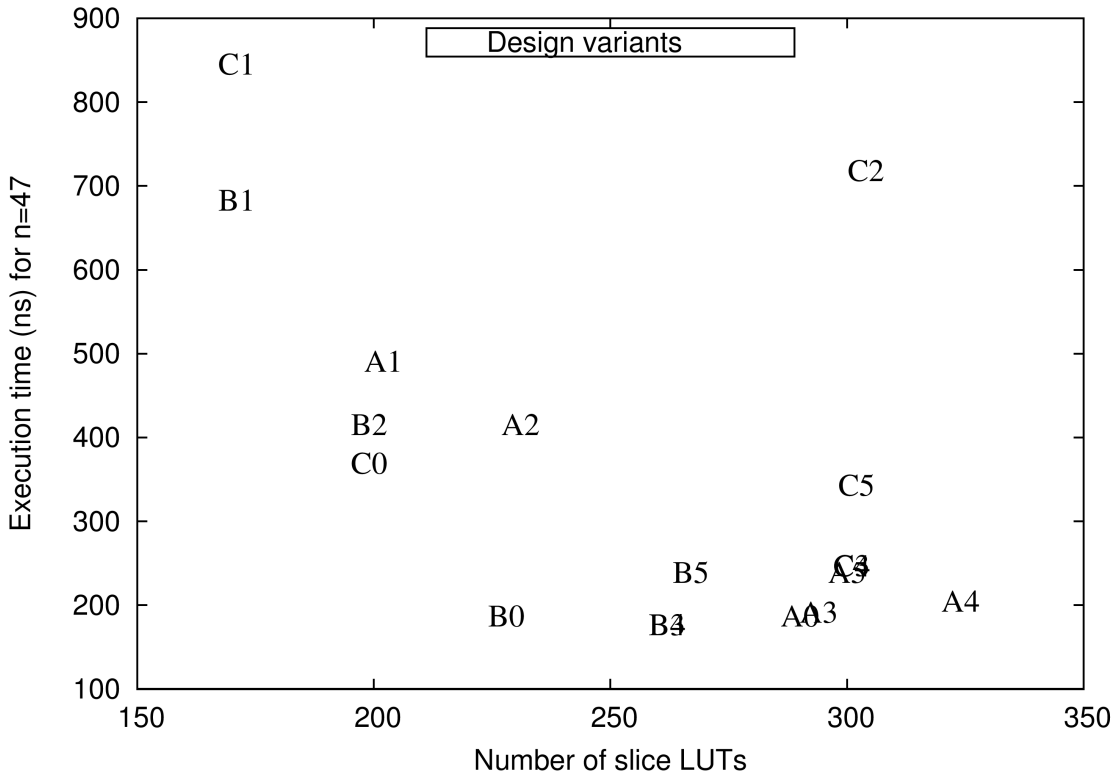
C code

# Fibonacci series example: Machine cycles

| Design | Cycles | Design | Cycles | Design | Cycles |
|---|---|---|---|---|---|
| **A0** | $n$ | **B0** | $n$ | **C0** | $2n-1$ |
| **A1** | $4n+3$ | **B1** | $5n+2$ | **C1** | $7n+1$ |
| **A2** | $4n+2$ | **B2** | $4n+2$ | **C2** | $7n$ |
| **A3-A5** | $n+2$ | **B3-B5** | $n+2$ | **C3-C5** | $2(n+1)$ |

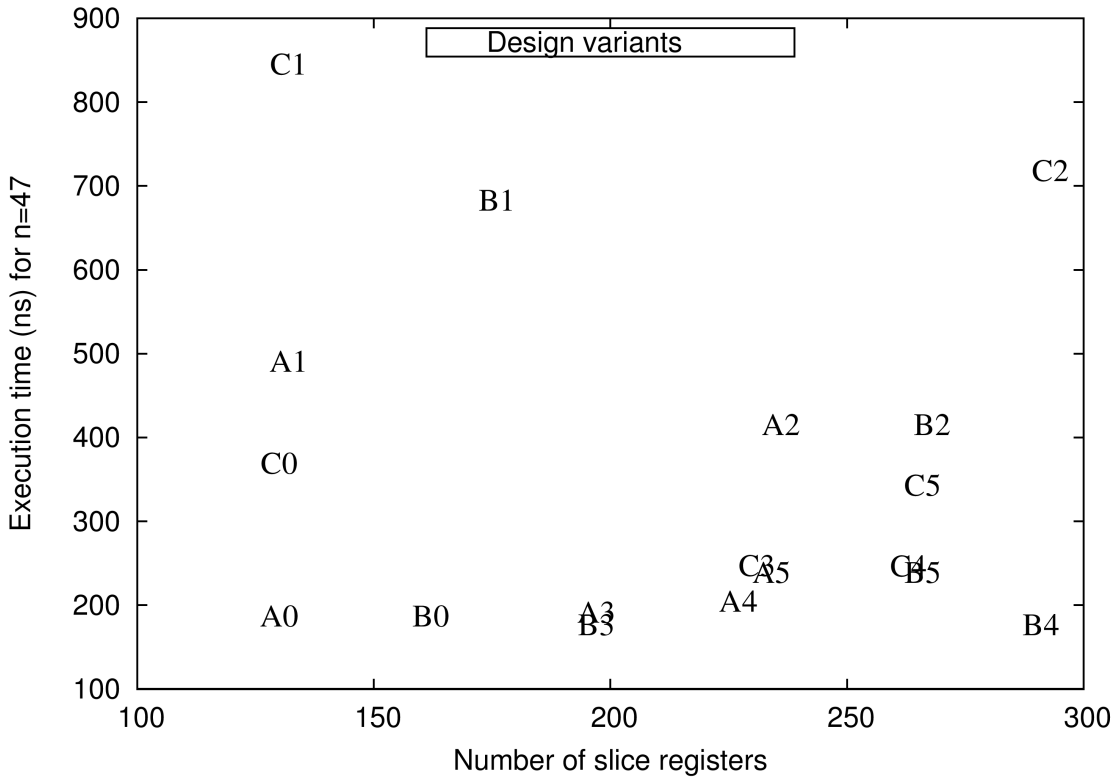
- Hand-optimized designs are A0, B0, C0
- The benefit of ASAP is not significant due to the data dependencies in the algorithm
- Cycle reduction is achieved through operation chaining
- HercuLeS can closely match the result of a human expert for optimization schemes O3-O5
- The slight differences in cycle performance are due to specific design choices of the human expert
  - initializing $f0$, $f1$ and $k$ in the FSMD entry state
  - passing the output data argument without use of an intermediate register

# Fibonacci series example: Execution time vs LUTs



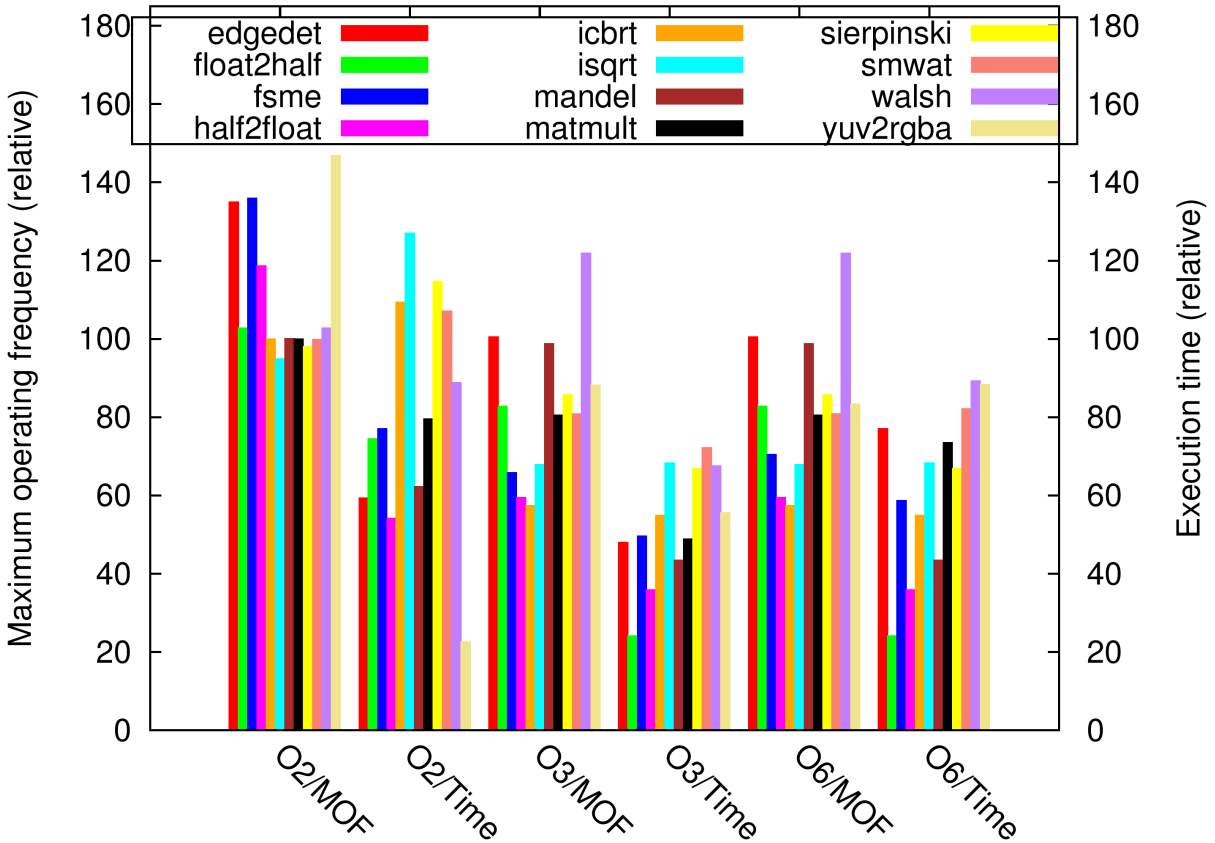

- Better results are placed near the bottom-left corner
- Pareto-optimal designs by human expert: B0 and C0
- Pareto-optimal designs by HercuLeS: B2, B3, B4

# Fibonacci series example: Execution time vs Registers



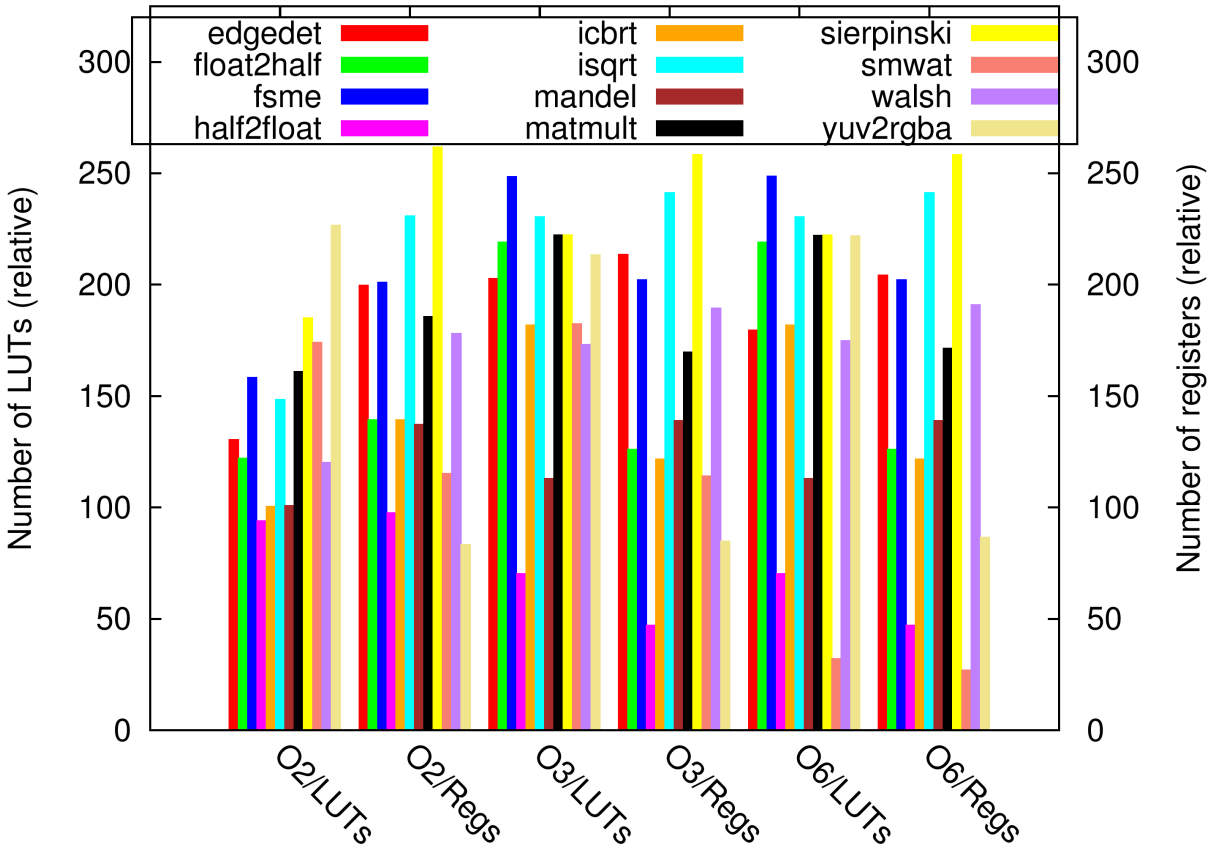

- Better results are placed near the bottom-left corner
- Pareto-optimal designs by human expert: A0
- Pareto-optimal designs by HercuLeS: A1 and B3

# Benchmark results: Speed




- Average computation time is reduced by 44.3% when comparing O1 to O3
- This gain is limited to 37.3% for O6 due to block RAM timing
- *float2half* and *half2float* achieve up to 4× execution time reduction
- Maximum operating frequencies in the range of 119-450MHz

# Benchmark results: Area




- O1 generates (slower and) smaller hardware in terms of LUTs and registers
- O3 introduces the highest LUT requirements; O2 the highest register demand
- Registers are reduced by 17.5% among O2 and O6; LUTs by 14% among O3 and O6
- Block RAM inference leads to significant LUT/register area reduction (*smwat*)

# QoR against commercial competition

- New frontends, analyses and optimizations are easy to add
- Maps character I/O and `malloc`/`free` to efficient hardware
- Uses open specifications and formats (NAC, Graphviz)
- Vendor and technology-independent HDL code generation
- Preliminary results against Vivado HLS 2013.1

| Benchmark | Vivado HLS | | | HercuLeS | | |
|---|---|---|---|---|---|---|
| | LUTs | Regs | Time (ns) | LUTs | Regs | Time (ns) |
| Array sum | **102** | 132 | **26.5** | 103 | **63** | 73.3 |
| Bit reversal | 67 | **39** | 72.0 | **42** | 40 | **11.6** |
| Edge detection∗ | **246** | **130** | 1636.3 | 680 | 361 | **1606.4** |
| Fibonacci series | 138 | **131** | **60.2** | **137** | 197 | 102.7 |
| FIR filter | **102** | **52** | **833.4** | 217 | 140 | 2729.4 |
| Greatest common divisor | 210 | 98 | **35.2** | **128** | **93** | 75.9 |
| Cubic root approximation | **239** | 207 | **260.6** | 365 | **201** | 400.5 |
| Population count | **45** | **65** | **19.4** | 53 | 102 | 26.1 |
| Prime sieve∗ | 525 | 595 | 6108.4 | **565** | **523** | **3869.5** |
| Sierpinski triangle | **88** | **163** | **11326.5** | 230 | 200 | 16224.9 |

# Summary

- HercuLeS is an extensible HLS environment for hardware and software engineers
  - Straightforward to use from ANSI C, generic assembly (NAC) or custom DSLs (Domain Specific Languages) to producing compact VHDL designs with competitive QoR
  - Users can investigate different optimization scenarios regarding speed and area
  - Product information
    - HercuLeS GUI for specifying code generation, simulation and synthesis options
    - Commercial distribution: `http://www.ajaxcompilers.com`
    - Technical details: `http://www.nkavvadias.com/hercules`
    - **FREE**, **BASIC**, and **FULL** software licensing schemes (2013.a version)

# Thank you

for your interest