POLITECNICO DI MILANO

Design, Automation and Test in Europe Conference
March 21, 2022

Modern High-Level Synthesis for Complex Data Science Applications

Compiler Based Optimizations, Tuning and Customization of Generated Accelerators

Michele Fiorito

Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
michele.fiorito@polimi.it

Outline

- Accelerator interface
- Front-end target-independent optimizations
- Middle-end hardware-oriented optimizations
- Bambu HLS algorithms
- Integer and floating-point math support

Reset signal

- Internal status of accelerator can be reset
  - Accelerator exposes a reset signal
- Register reset type:
  - no (default)
  - async
  - sync
- Reset level:
  - low (default)
  - high
- Example:

```
--reset-type=sync --reset-level=high
```

Registered inputs

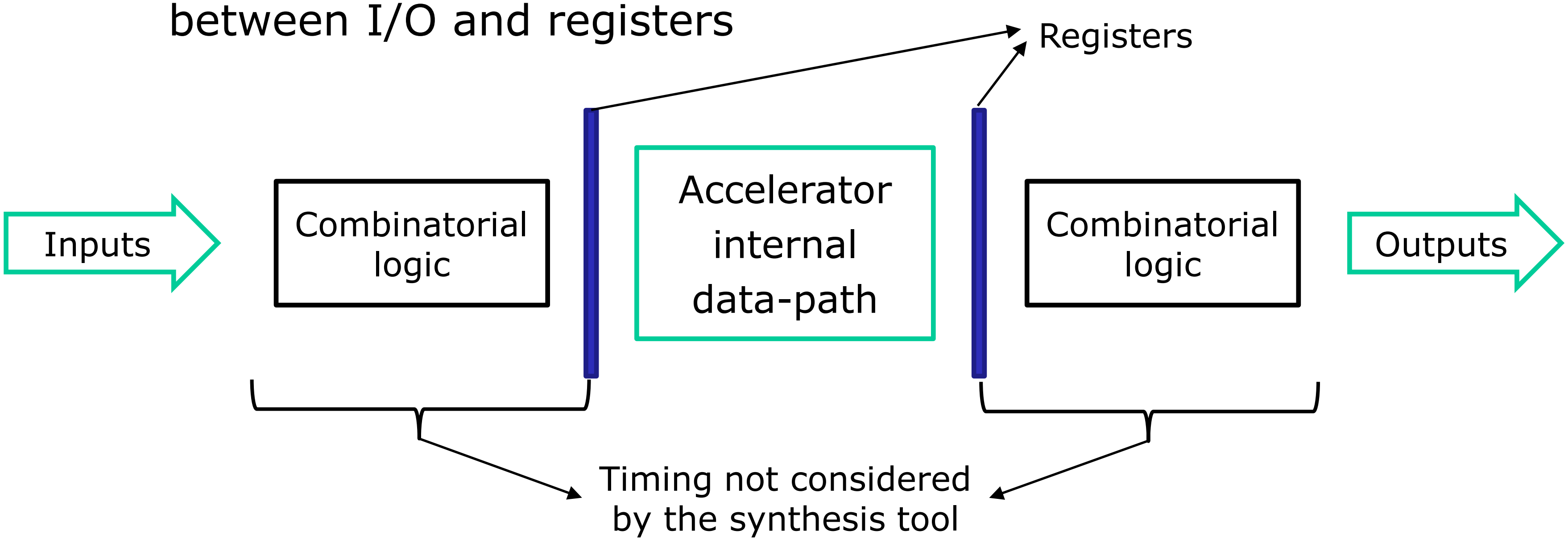
- A dedicated port is created for scalar parameters of each module function
- Generated modules expect stable inputs
  - If inputs are not stable, they can be registered
- Registered inputs:
  - **auto** – (default) inputs are registered only for shared functions
  - **top** – inputs are registered for top interface and shared functions
  - **yes**
  - **no**

```
--registered-inputs=<value>
```

Registered inputs

Unregistered top-level interface:

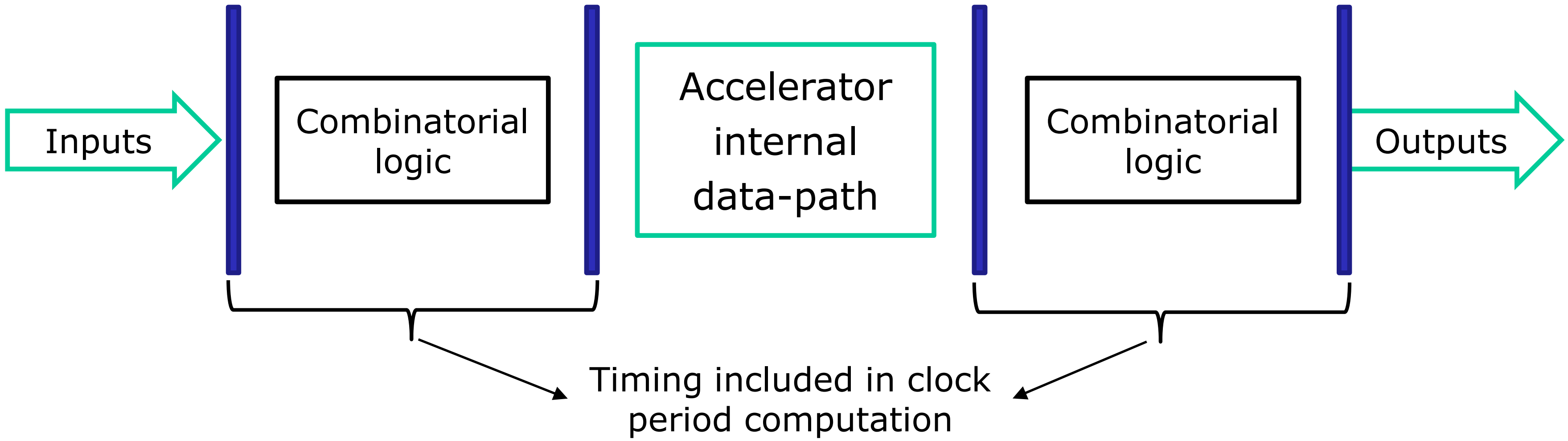
- ❑ Unfaithful global timing computation
- ❑ Scheduling suffers from unregistered logic
- ❑ Inaccurate timing estimation of logic between I/O and registers



Registered inputs

Registered top-level interface:

- Accurate timing estimation
- Better operations' scheduling
- More stable interface



FSM-encoding

- Different types of encoding can be used in Finite State Machine
  - one-hot (default) – best for performance
  - **binary** – best for area

```
--fsm-encoding=<value>
```

Accelerator design optimizations

- Performance and area of the generated accelerators can be improved by tuning the design flow
  - GCC/CLANG target-independent optimizations
  - Bambu IR hardware-oriented optimizations
  - HLS algorithms (allocation, scheduling, binding)
- Best design flow for all accelerators does not exist
  - Trade off between area and performance
  - Effects of the single optimizations can be different on different input applications
- Default optimization flow:
  - Balanced area/performance trade-off

GCC/CLANG optimizations

- Target-independent optimizations only
- User can tune this part of the flow:
  - Selecting optimization level:

```
-O0 or -O1 or -O2 or -O3 or -Os
```

  - Enabling/disabling single optimization:

```
-f<optimization> -fno-<optimization>
```

  - Tuning parameters: --param

```
--param <name>=<value>
```

Impact of global optimization level

- Bambu defaults are used changing front-end compiler optimization level only

Optimization level	Clock cycles	LUTs
O0	15764	11675
O1	7892	11052
O2	4679	10276
O3	3854	15679
O3 vectorize	3816	38553
O3 all inline	1327	13550

- -O3 is not necessarily the best choice
  - Can improve **performances**
  - Can increment **area**

Bambu IR Optimizations

- Hardware-oriented optimizations

- Many optimization techniques:
  - Single instruction optimizations
  - Multiple instruction optimizations
  - Restructuring of Control Flow Graph
  - Rewriting IR

- Same optimization can be applied many times
  - Fixed point iteration optimization flow

Bambu IR Analysis

- Collects information over IR to be used by othe optimizations and HLS back-end
- Data flow analysis
  - Scalar: based on SSA
  - Aggregates (i.e. Front-end+Bambu alias analysis)
- Graphs Computation
  - Call Graph, CFG, DFG, …
- Loops identification
- Bit Value Analysis
  - Compute for each SSA variable which bit are used, which are fixed, which are useless
- Range Analysis

Single Instruction Optimization

- IR lowering – make single instructions more suitable to be implemented on FPGA
  - Expansion of multiplication by constant
  - Expansion of division by constant
  - Etc.

- Bit Value Optimization – exploit information from previous IR analyses to make bitwise optimizations
  - Shrink operations to the only significant bits

Multiple Instruction Optimization

- Common Subexpression Elimination
- Dead Code Elimination
- Extract pattern (e.g., three input sum)
- LUT transformations
  - Merging multiple Boolean operations into a single LUT-based operation
- Conditional Expression Restructuring
- Commutative Expression Restructuring

Restructuring of Control Flow Graph

- **Merging** of conditional branch
  - Creation of multiple target branch
- **Basic Block Manipulation**
  - Remove (empty, dead, …)
  - Split
  - Merge
- **Code motion**
- **Speculation**

Rewriting IR

- Struct assignment
  - Replaced with memcpy call
- Floating point operations
  - Replaced with function calls
- Integer divisions
  - Replaced with function calls

System of Difference Constraints

- Global scheduling based on ILP formulation
- Results are exploited to perform
  - Speculation
  - Code Motion

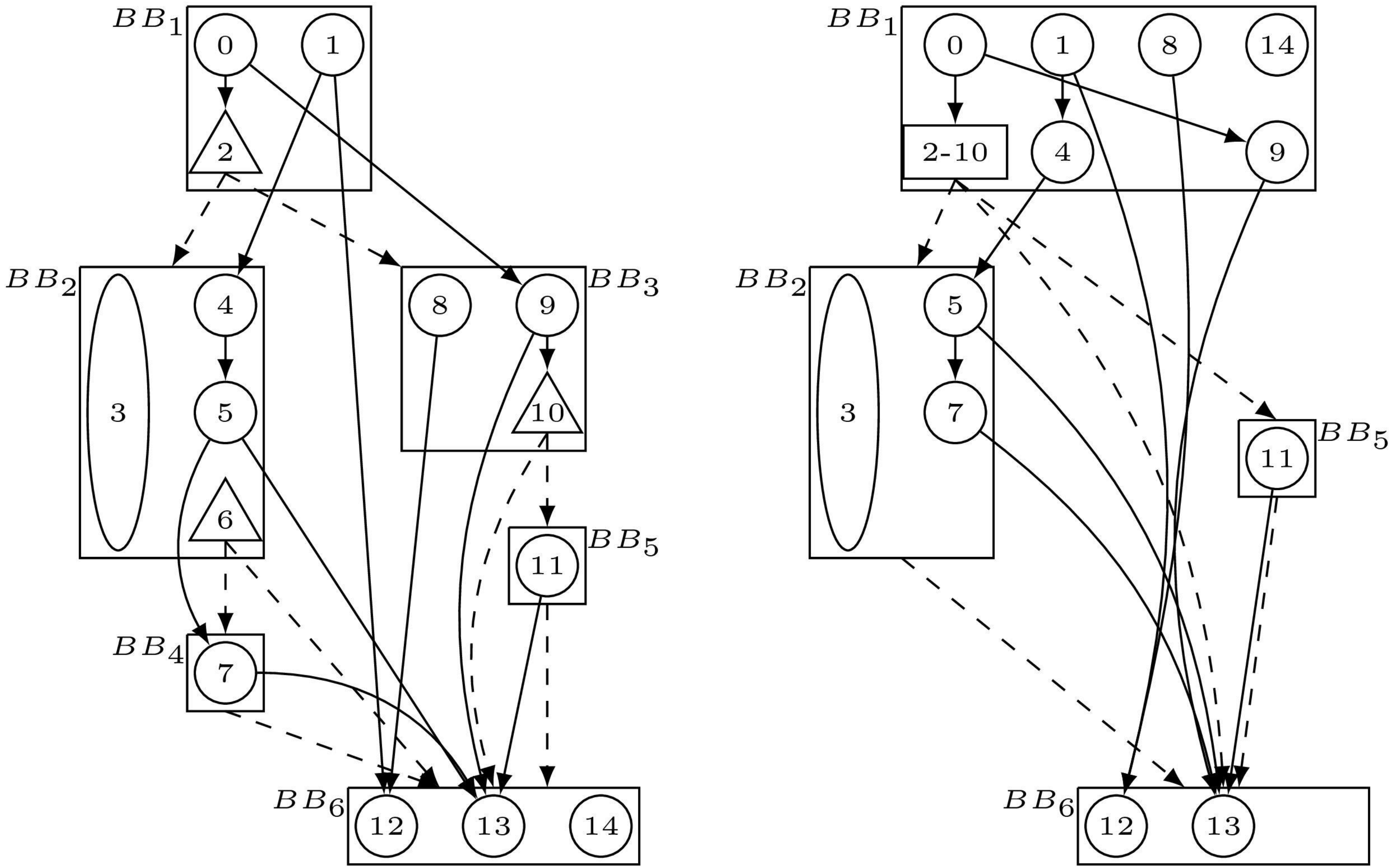
+ Improve performances of accelerators

- Potentially increment area of accelerators

- Increase High Level Synthesis time

```
--speculative-sdc-scheduling
```

Example of scheduling optimization



Experimental setups

- Predefined optimizations' set

```
--experimental-setup=<setup>
```

BAMBU-AREA: optimized for area

BAMBU-PERFORMANCE: optimized for performances

BAMBU-BALANCED: optimized for trade-off area/performance

BAMBU-AREA-MP, BAMBU-PERFORMANCE-MP, BAMBU-BALANCED-MP: enable support to true dual port memories

Default: BAMBU-BALANCED-MP

Resource Constraints

- Bambu assumes infinite resources on target
  - Produced solutions may not fit in the target device

- Area of generated solutions can be indirectly controlled by means of constraints

- Function-scope constraints on number of functional units
  - E.g.: fix the number of available multiplier in each function

- Constraints are set by means of *XML file*

Example of constraints file


```xml
<?xml version="1.0"?>
<constraints>
  <HLS_constraints>
    <tech_constraints fu_name="mult_expr_FU"
                      fu_library="STD_FU" n="8" />
  </HLS_constraints>
</constraints>
```

Integer Division Algorithms

- User can control integer division implementation:

```
--hls-div=<implementation>
```

- Available implementations:
  - `none`: HDL-based pipelined restoring division
  - **`nr1`** (default): C-based non-restoring division with unrolling factor equal to 1
  - `nr2`: C-based non-restoring division with unrolling factor equal to 2
  - `NR`: C-based Newton-Raphson division
  - `as`: C-based align divisor shift dividend method

Floating point support

- Possible ways of implementing floating point ops:
  - Softfloat (default): customized faithfully rounded (nearest even) implementation

    `--soft-float`

  - Subnormals: subnormal numbers support can be enabled through

    `--fp-subnormal`

  - Softfloat GCC: GCC soft-based implementation

    `--soft-fp`

  - FloPoCo generated VHDL modules

    `--flopoco`

Libm versions

- HLS flow exploited to generate hardware implementation of soft-defined libm functions

- Two different versions of libm are available
  1. Faithfully rounded libm (default)
  2. Classical libm built integrating existing libm source code from glibc, newlib, uclibc and musl libraries.
     - Worse performances and area

Hands-on time

Switch to Colab Notebook to test some of bambu optimizations

First example – ADPCM

Benchmark	CYCLES	HLS_execution_time
GCC49:adpcm_O0	33429	23,05
GCC49:adpcm_O1	24547	18,72
GCC49:adpcm_O2	24043	43,26
GCC49:adpcm_O3	10429	76,45
GCC49:adpcm_O3_inline	7503	99,58
GCC49:adpcm_O3_vectorize	6995	49,31
GCC49:adpcm_Os	24847	25,21

Second example – ADPCM

Benchmark	CYCLES	HLS_execution_time
GCC49:adpcm_O0_sdc	33479	64,38
GCC49:adpcm_O1_sdc	24297	57,09
GCC49:adpcm_O2_sdc	22863	83,53
GCC49:adpcm_O3_sdc	9149	175,93
GCC49:adpcm_O3_inline_sdc	5356	210,62
GCC49:adpcm_O3_vectorize_sdc	6135	110,81
GCC49:adpcm_Os_sdc	24397	68,45

Third example – Integer Division

Benchmark	CYCLES	HLS_execution_time
GCC49:dfdiv_none	1777	37,5
GCC49:dfdiv_nr1	1849	41,18
GCC49:dfdiv_nr2	1105	43,12
GCC49:dfdiv_NR	825	44,92
GCC49:dfdiv_as	841	30,14

Question time

Research supported by HERMES project
HERMES project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 101004203