Generating Hardware Description with Target-Independent Code Generator

Zheng, Hongbin
etherzhhb@gmail.com

Hello I am Zheng, Hongbin today I will introduce our LLVM based HLS framework, which is built upon the Target-Independent Code Generator.

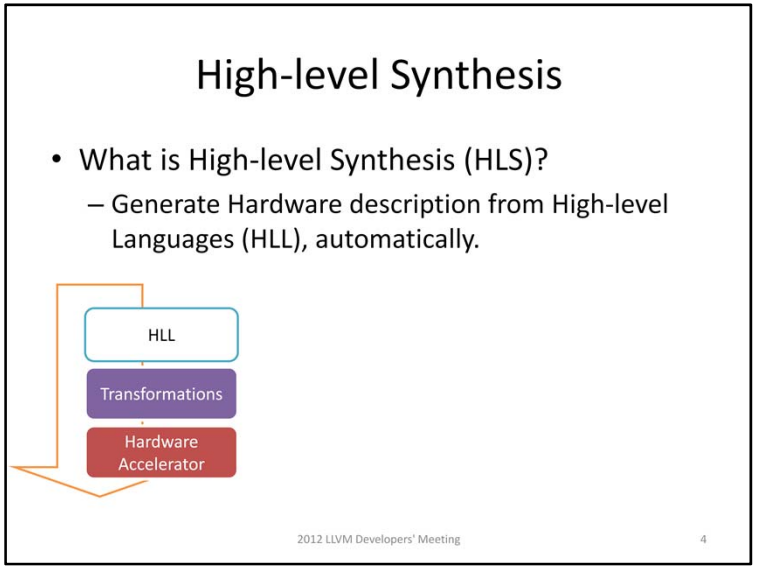
Outline

- Introduction
- Overview of the Shang HLS framework
- The Cross-level Engine
- Kernel-only Software Pipelining
- Experimental Results and Further Work

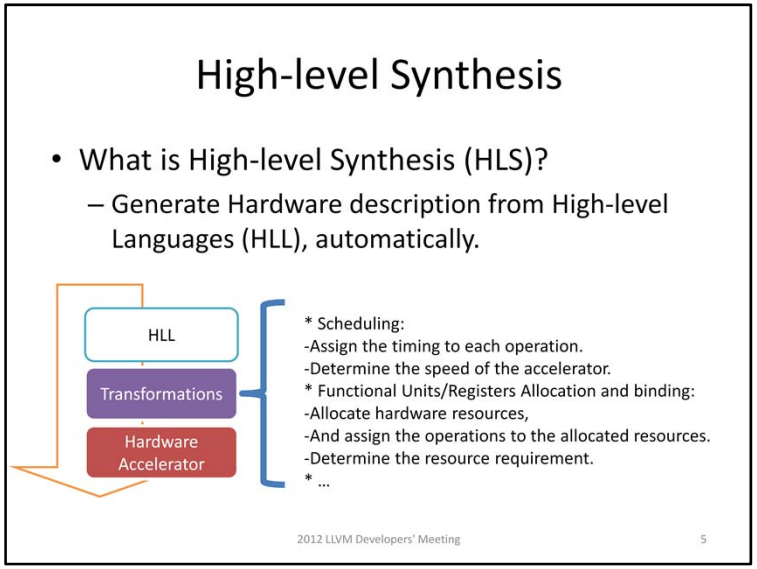
In this talk I will first briefly introduce what HLS is, and then given an overview of the HLS framework, named “Shang”.
After that I will provides some details of the HLS framework, including the Cross-level engine, and the Kernel-only Software Pipelining technique implemented in the framework.
At last, I will give the experimental result and the Further Work.

Outline

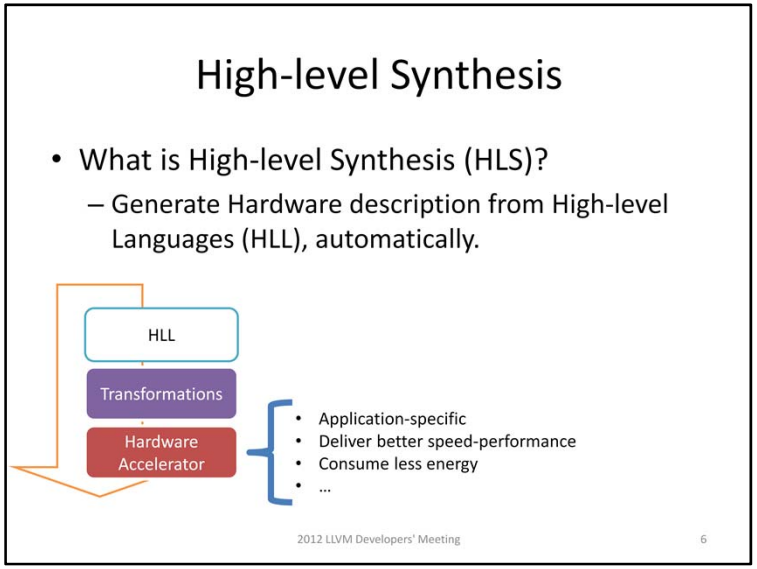
- Introduction
- Overview of the Shang HLS framework
- The Cross-level Engine
- Kernel-only Software Pipelining
- Experimental Results and Further Work



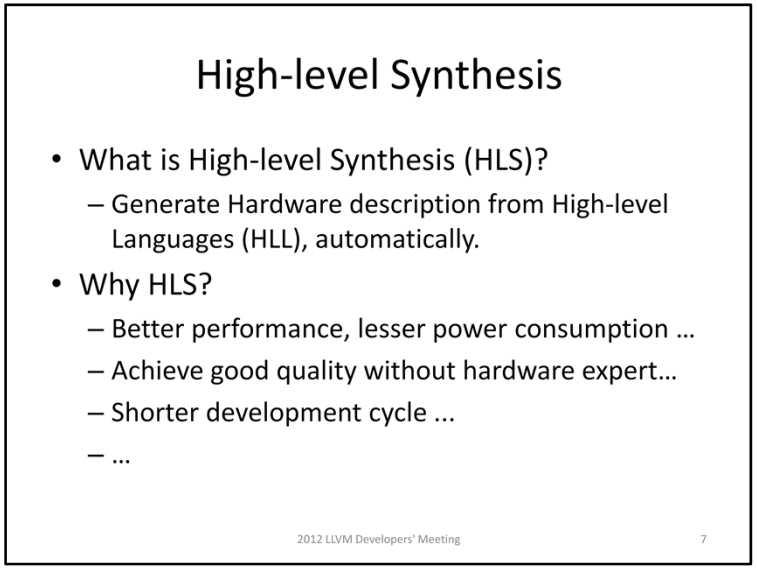
So first of all, some people may ask what is HLS?
HLS is a technique that automatically generate the Hardware Description of the hardware accelerator from High-level Language, which is designed for software development only, like C/C++, C# and Java.



During the transformation process, there are two key tasks should be perform:
That is Scheduling and Resource allocation and binding.
At this point, HLS share some similarity with software compilation process.



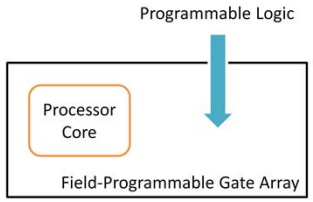
At last HLS generate the description for the hardware accelerator, which describe the structure and the timing of the circuit.
With the Hardware accelerator, we can achieve better speed-performance while consuming less energy,
at the cost of hardware resource usage and development time,
comparing to implementing the same functionality with general-propose processors.



So the advantages of the hardware accelerator may had already answer why we need HLS.
But that not the most important reason, because the hardware accelerator may be designed manually.
The may reason for using HLS is that, HLS can achieve good quality, or at least, not so bad quality with a hardware newbie, with a shorter development time than manually design.

HLS Example

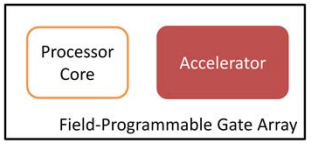
- There exist some FPGAs with ARM core:
  - Arria V SoC FPGA by Altera
  - Zynq by Altera Xilinx



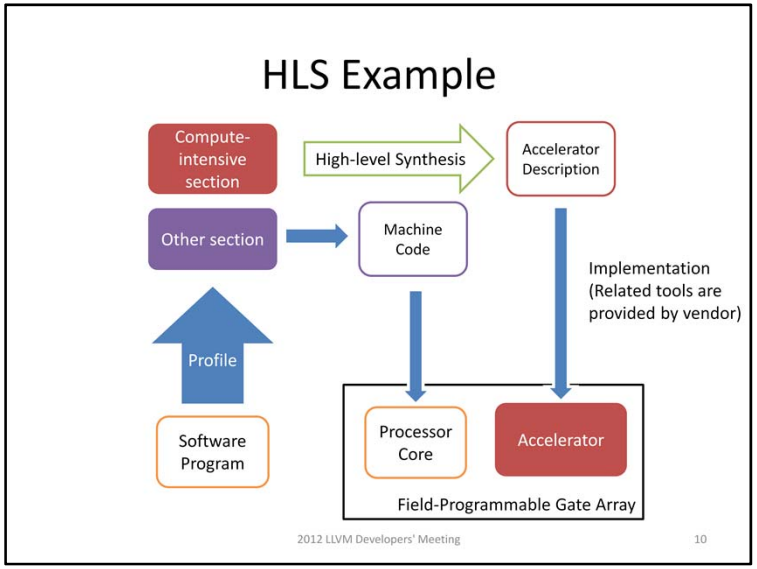
Here I can give an example of HLS.
Some people may know the some type of ARM processor core had been integrated with the some reconfigurable device.

HLS Example

- There exist some FPGAs with ARM core:
  - Arria V SoC FPGA by Altera
  - Zynq by Altera Xilinx
- The programmable logic can implement the hardware accelerator.



Such device allow us run software programs on the ARM core while implementing the accelerator on the reconfigurable logic.



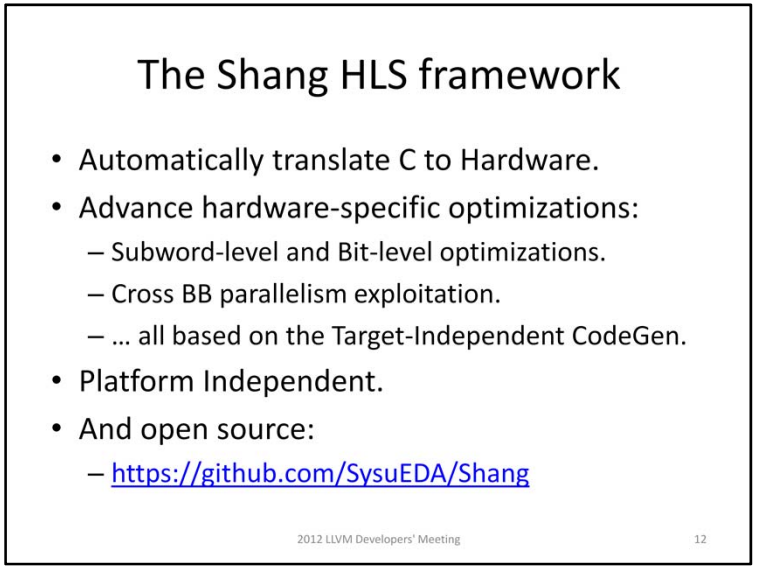
So given a program to run on these device, we can first profile the program to identify the “compute-intensive section” and then implement it as hardware accelerator, with the help of a HLS tool.
While letting the other part of the program run on the processor core.

Such design flow can leverage the computational power of the reconfigurable logic and the flexibility of the processor core.

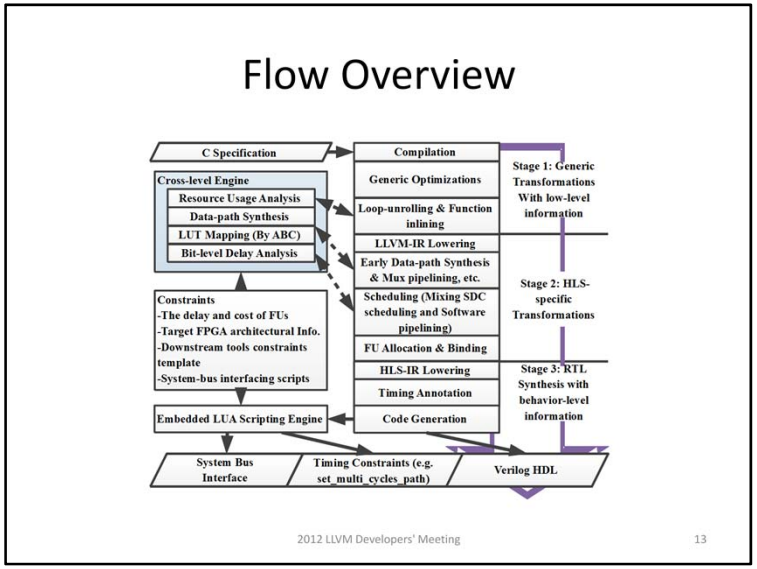
Outline

- Introduction
- Overview of the Shang HLS framework
- The Cross-level Engine
- Kernel-only Software Pipelining
- Experimental Results and Further Work

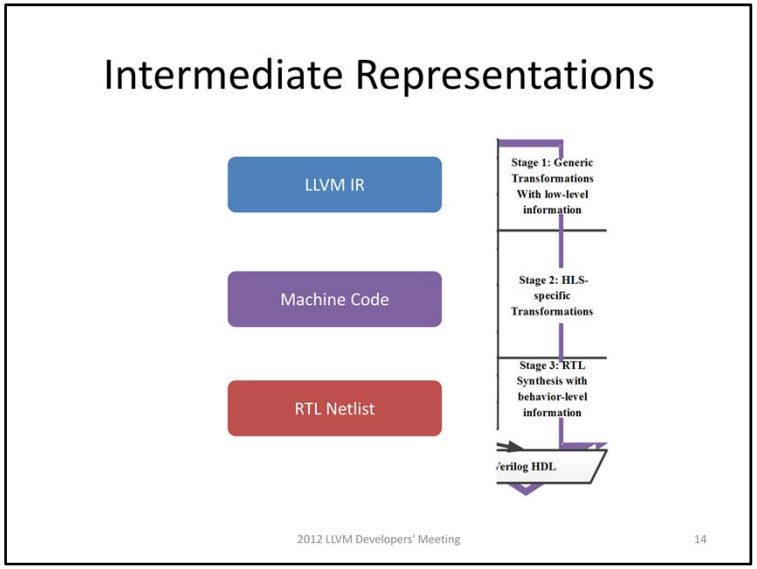
After we have an idea about HLS, I am going to introduce the Shang HLS framework.



The Shang HLS framework is developed as a research project.
It automatically translate C to Hardware, which some advance optimizations targeting the reconfigurable logic,
For example, it optimize the program at subword-level and bit-level, it support global code motion to exploit the parallelism beyond the BB boundaries.
In addition, the framework supports both Xilinx FPGA and Altera FPGA, with the help of the embedded scripting engine.
At last the framework is opensource.



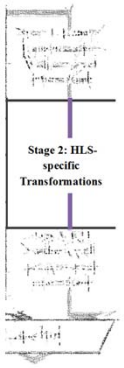
An overview of the HLS flow in the framework is given in this slide.
In general, the flow can be divided into three stages, each stage works at different abstract level.
At the same time, there is “Cross-level Engine” providing the low-level information to the high-level transformations.
In addition, the framework embedded an LUA scripting engine to read the platform-specific information and generate platform-specific constructs.



There is different kinds of IRs for each stages in the synthesis flow, including LLVM IR, Machine Code and RTL Netlist.

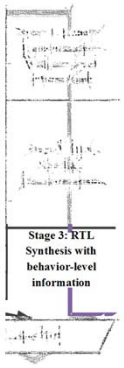
HLS-specific Machine Code

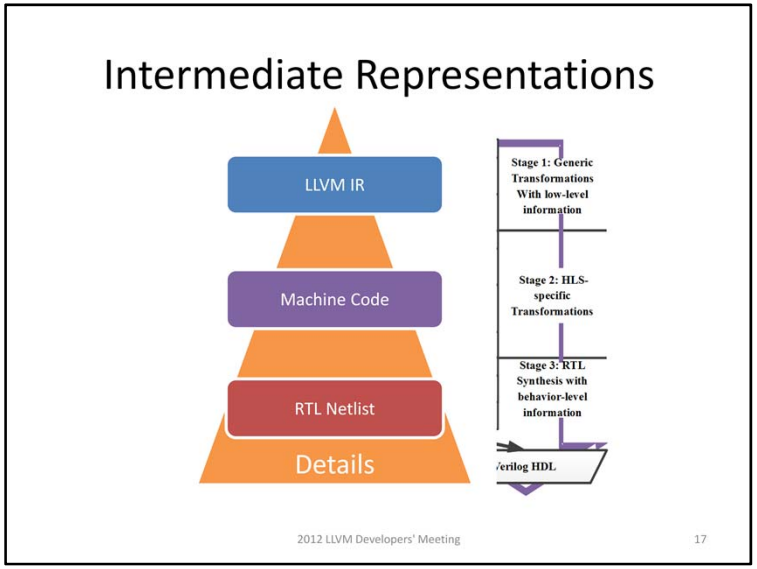
- A virtual instruction set targeting FPGA.
- Contains special instructions:
  - Bit concatenation, subword extraction, look-up table …
- All instructions are predicated.
  - Enable some advance control construct.
- Represents the behavior of the design.
  - In more details than LLVM IR.



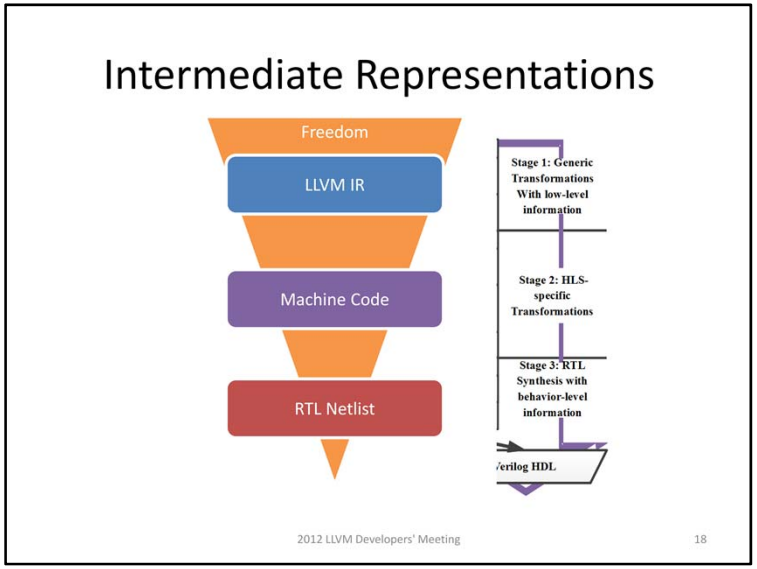
RTL netlist

- Derived from the scheduled and bound Machine Code.
- Explicitly describe the HW:
  - The Functional Units,
  - And how they are connected.
- The data transactions on the registers:
  - At which cycle?
  - Under what condition?

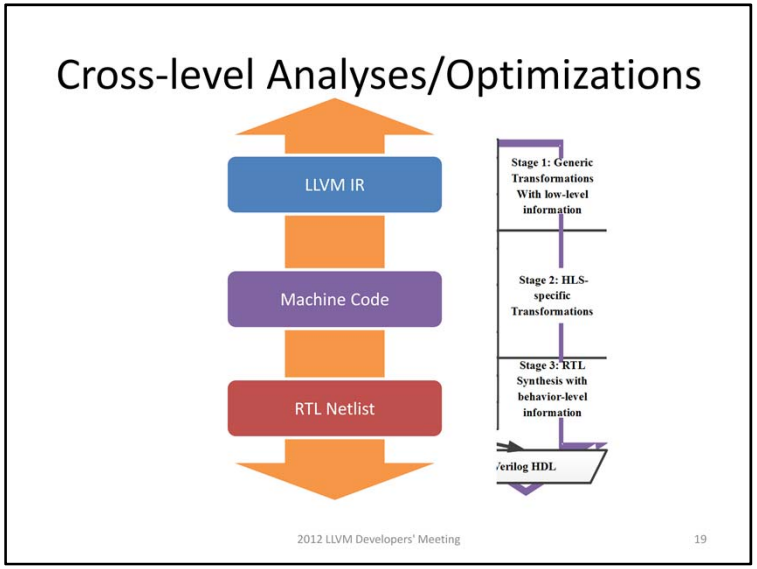




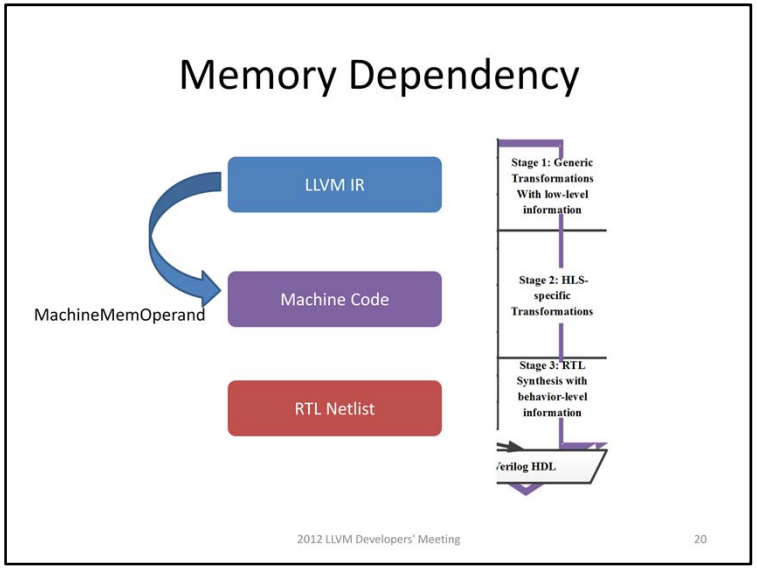
The closer to the end of the low the IR located, the more details are exposed.



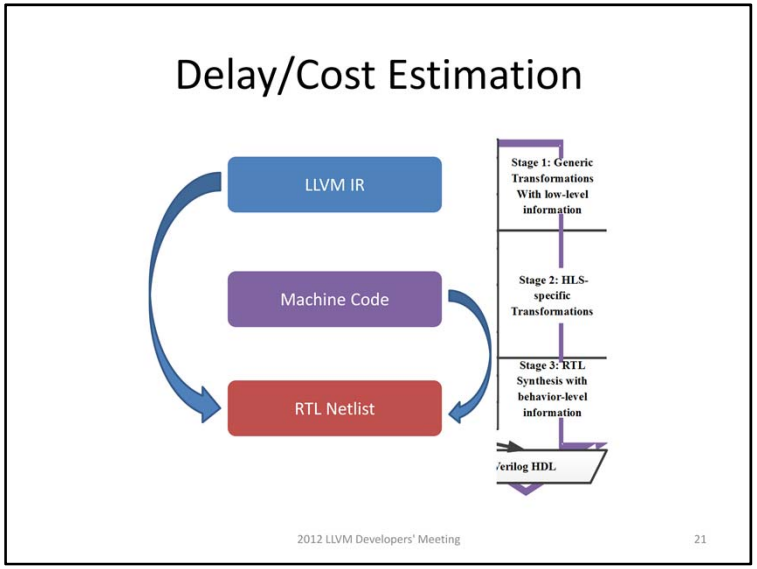
But the freedom of transformations is also reduced at the low level, because they need to preserve the constraints inherit from the High-level IRs.



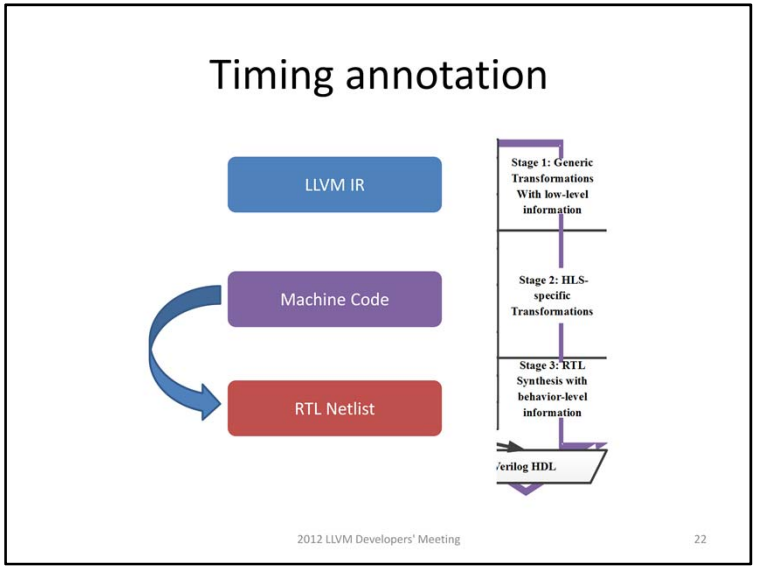
At the same time, cross reference is supported.



For example, the scheduler can query the memory dependencies with the MachineMemOperand.



Delay and cost estimation at high-level is provided by the cross-level engine.

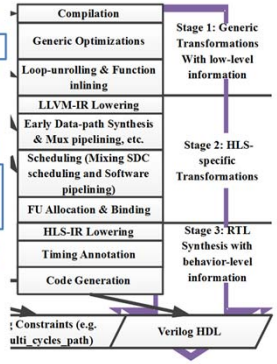


The timing information will also be annotated to the RTL Netlist, which means retiming with scheduling result is possible.

Flow implementation

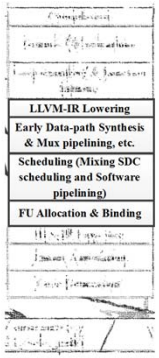
LLVM IR Passes

MachineFunction Passes, Based on a HLS-specific virtual instruction set.



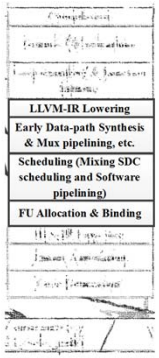
Fitting HLS flow into CodeGen

- Scheduling
  - Build the Scheduling Graph based on pre-register-allocation Machine code.



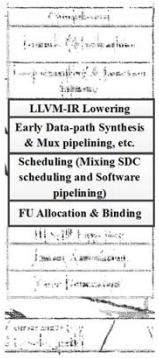
Fitting HLS flow into CodeGen

- Scheduling
  - Build the Scheduling Graph based on pre-register-allocation Machine code.
  - Pack instructions into bundles according to the scheduling results.



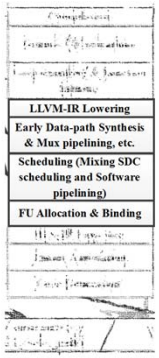
Fitting HLS flow into CodeGen

- Binding
  - Model all resource with physical registers.



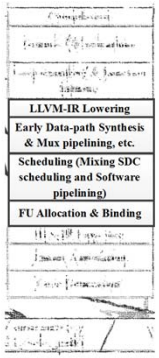
Fitting HLS flow into CodeGen

- Binding
  - Model all resource with physical registers.
  - LLVM helps to eliminate the PHIs.



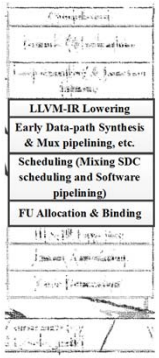
Fitting HLS flow into CodeGen

- Binding
  - Model all resource with physical registers.
  - LLVM helps to eliminate the PHIs.
  - LLVM helps to build the live-interval.



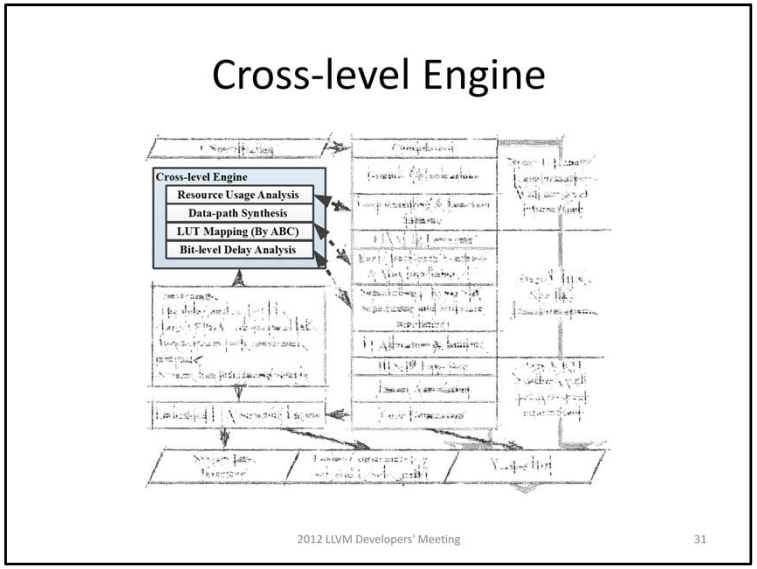
Fitting HLS flow into CodeGen

- Binding
  - Model all resource with physical registers.
  - LLVM helps to eliminate the PHIs.
  - LLVM helps to build the live-interval.
  - Perform LLVM physical register binding to solve the problem.

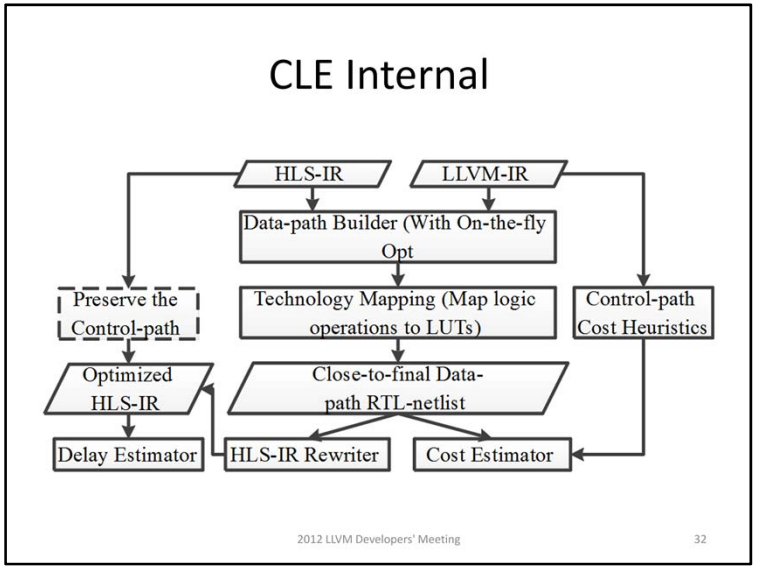


Outline

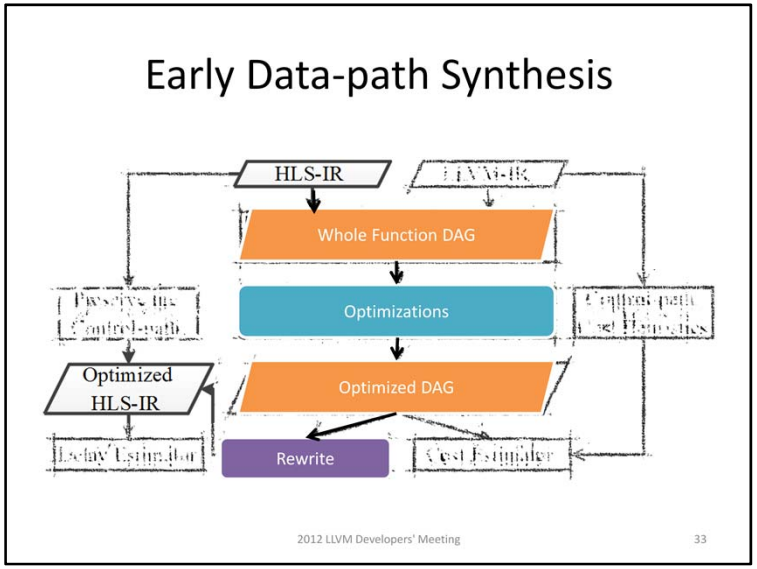
- Introduction
- Overview of the Shang HLS framework
- The Cross-level Engine
- Kernel-only Software Pipelining
- Experimental Results and Further Work



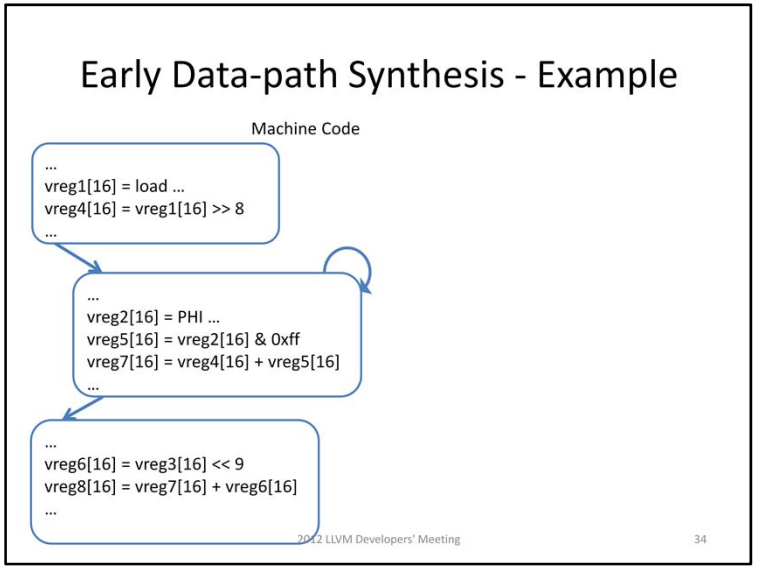
The cross-level engine provides the low-level information about the data-path, that is the computational part of the accelerator, to high-level transformations.



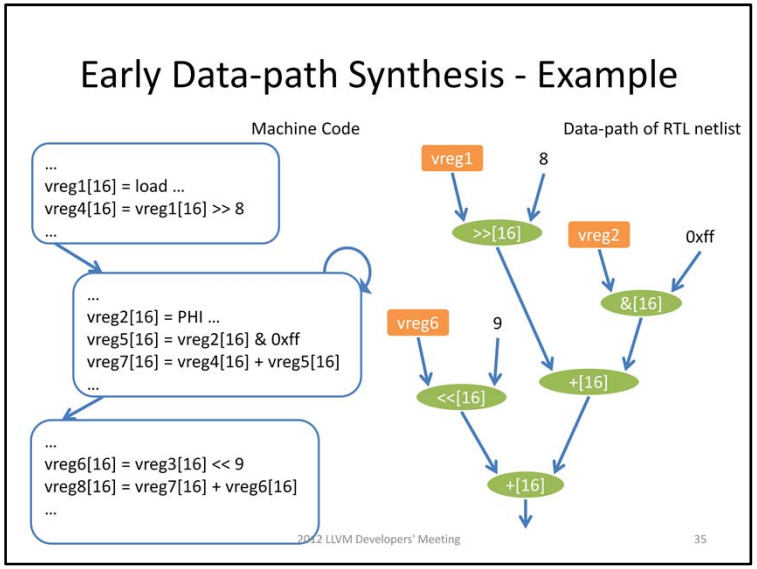
Specifically, it first build a close-to-final data-path netlist with bit-level/subword-level optimiations and Technology mapping. With the close-to-final netlist, we can either rewrite it back to high-level IR, or estimate its implementation cost.



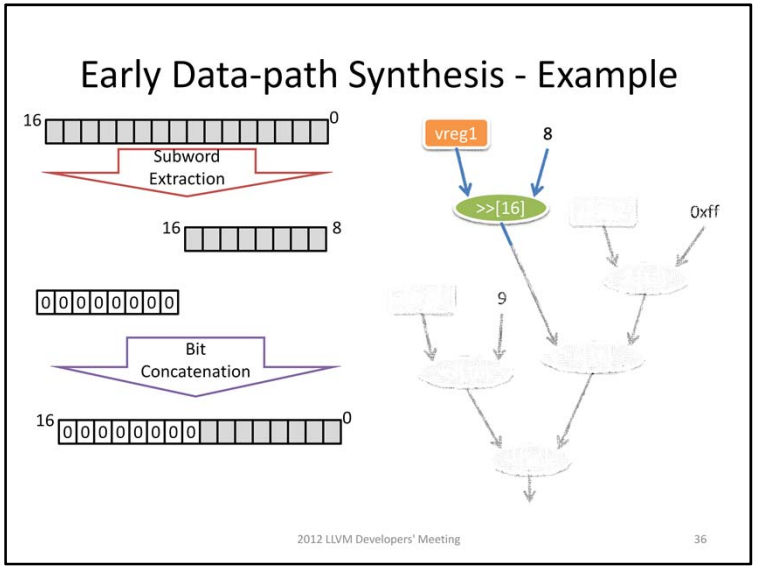
We call the bit-level/subword-level optimiations and Technology mapping in the cross-level engine as “Early Data-path Synthesis”.
I will give an example of the “Early Data-path Synthesis” on the machine code before scheduling and binding.
Essentially, it builds an whole function DAG, perform optimize on it and rewrite it back.



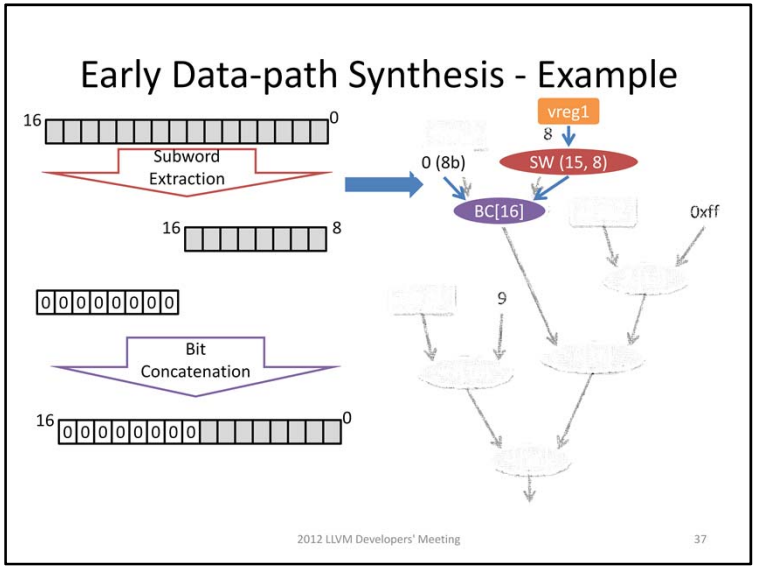
Supposed we started from this piece of machine code.
It contains addition, bit-wise and and shift. As well as load and PHIs



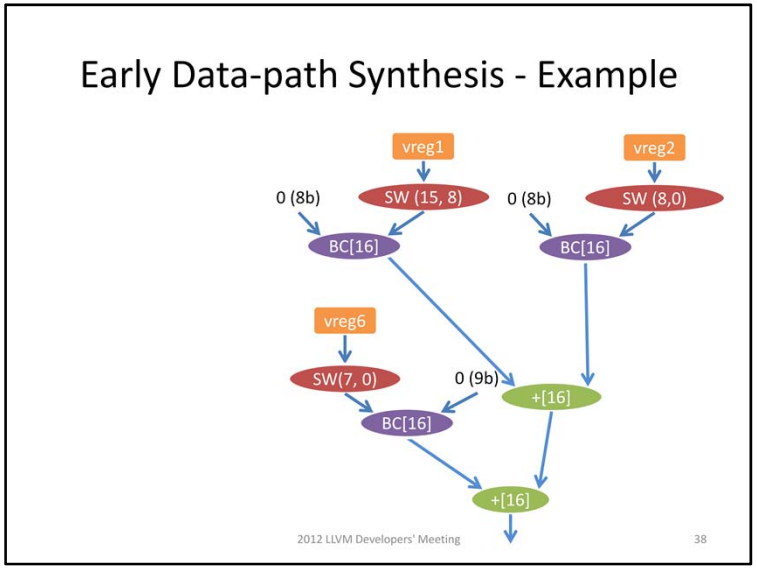
First of all, a DAG is built according to the Machine Code.
the instruction that cannot be optimized, e.g. load and PHI, are treated as unknown nodes, like SCEV.
In fact the Early Data-path Synthesis is somehow like SCEV, except it can also perform HLS-specific optimizations.



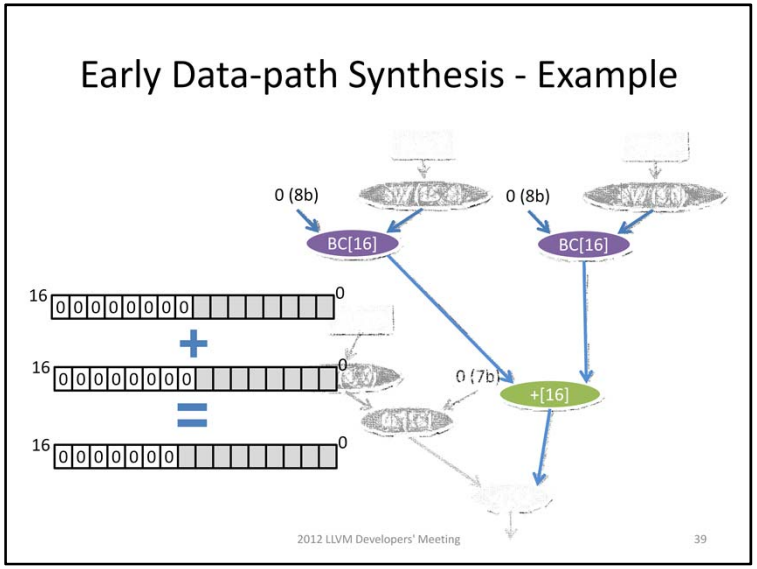
In the netlist, we may be able to identify some optimization opportunities.
For example, logical-shifted by constant amount can be replaced by subword extraction and bit concatenation.



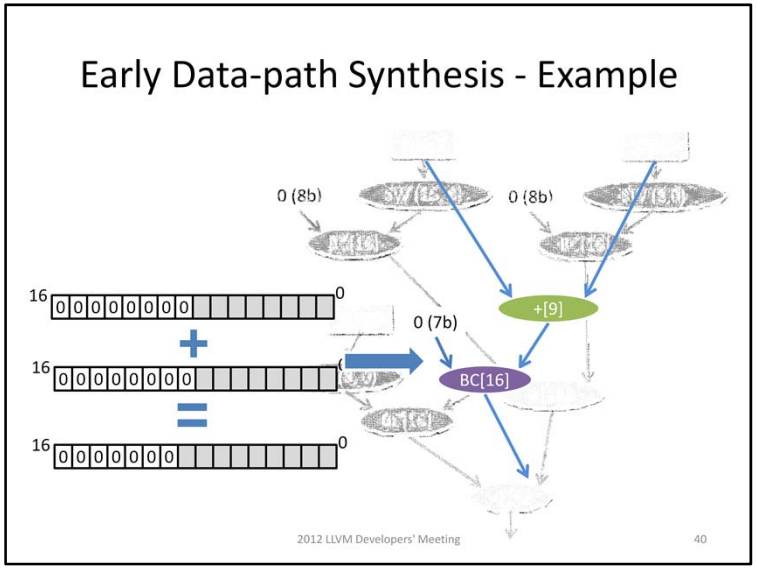
Hence we can rewrite the shift like this.



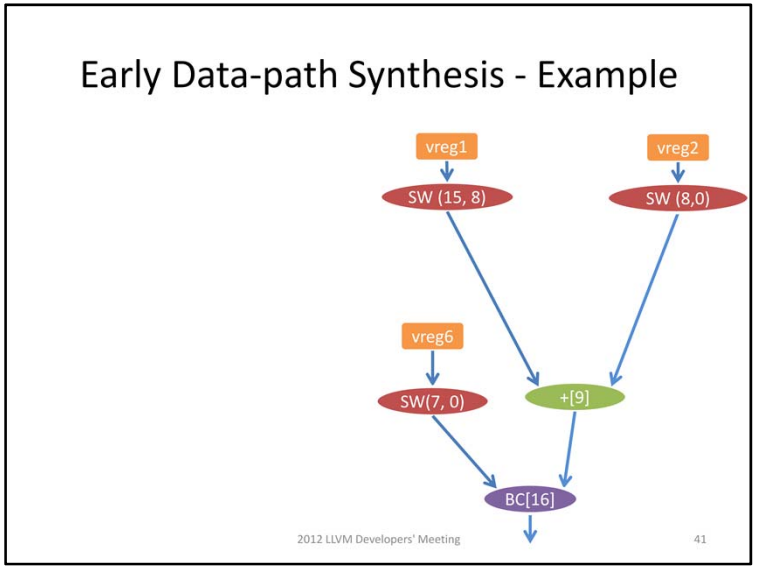
By applying similar optimizations, we can obtain a netlist like this, we replaced nodes by subword-extraction and bit-concatenation whenever possible, because these two operations are zero cost in the hardware accelerator.



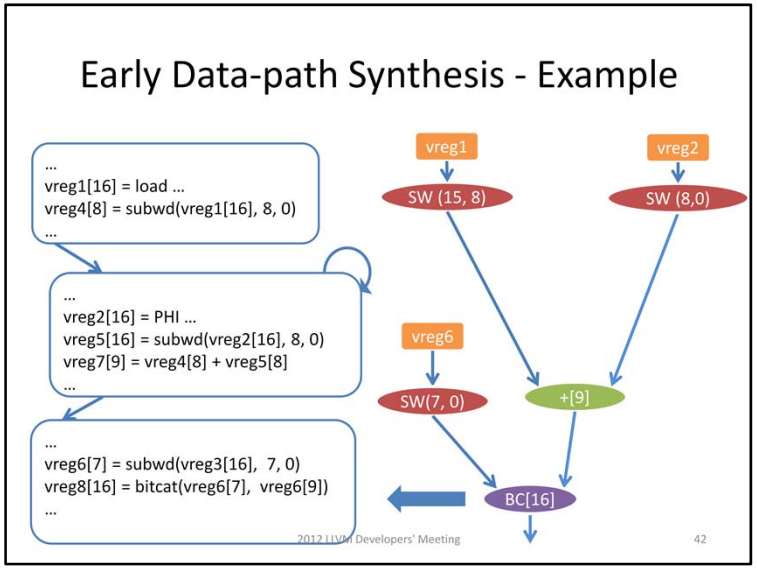
At the same time, bit-concatenation can provides the bit-mask information explicitly.
Hence we could take the advantage of the bit-mask to optimize the arithmetic operations.



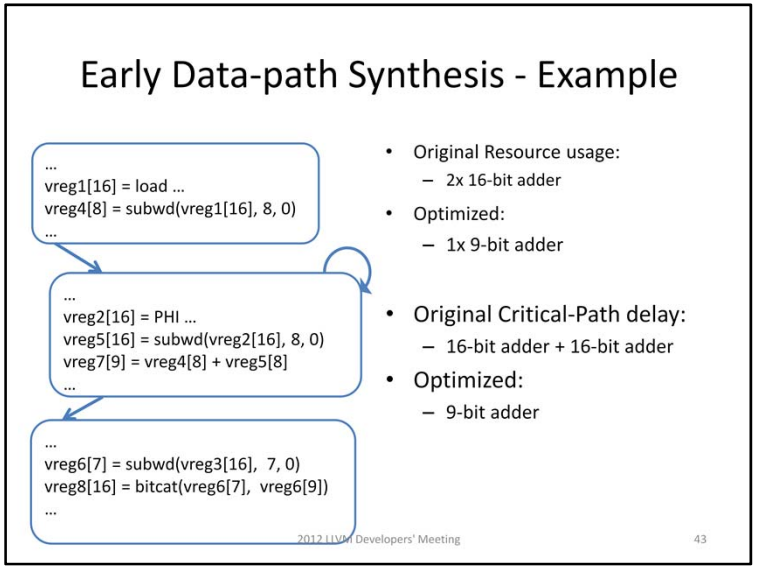
Hence, we can replace the 16-bit addition by a 8-bit addition and a bit-concatenation.



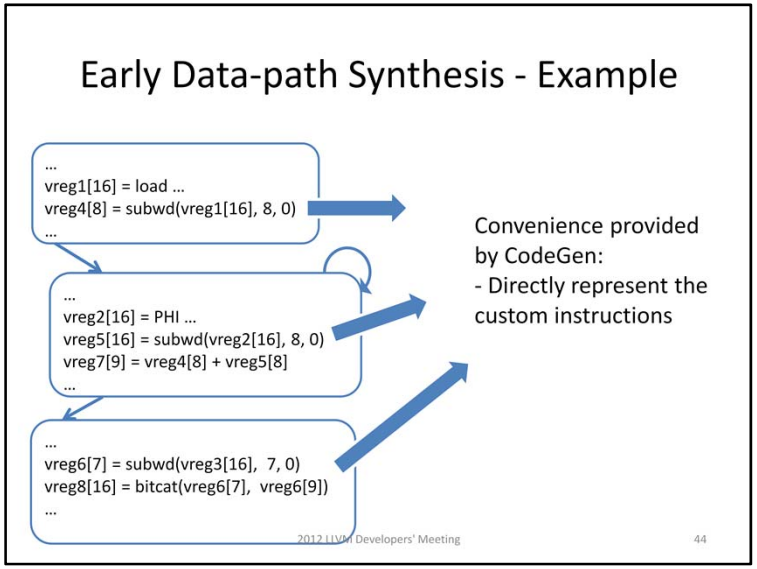
Finally we can get a netlike this, you can see that the operations are replaced by the ones that have a lower cost.



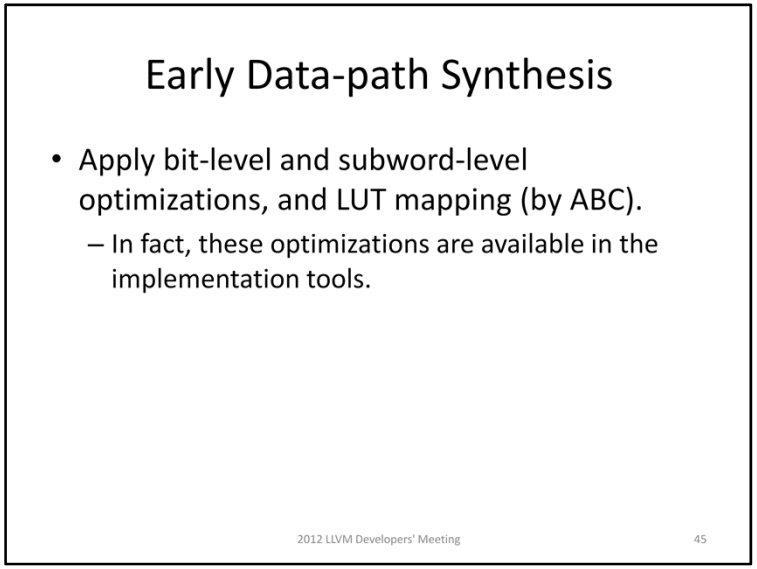
At last the Machine Code is rewritten according to the optimized netlist.



The rewritten netlist is more compact comparing with the original one.



Because the HLS-specific operations are directly expressed by the MachineInstructions.
We can directly analysis the rewritten Machine code to get the delay/cost estimation.



In summary, Early data-path synthesis can apply low-level operaitmiations.
But In fact, there are also available in the implementation tools which translate the description to implementations.

Early Data-path Synthesis

- Apply bit-level and subword-level optimizations, and LUT mapping (by ABC).
  - In fact, these optimizations are available in the vendor implementation tools.
- Transform the design representation toward final form.
  - Provide better delay estimation to the scheduler.
  - Provide better cost estimation to the binder.

However, doing this early could provide more accurate estimation to the scheduler and the binder.

Outline

- Introduction
- Overview of the Shang HLS framework
- The Cross-level Engine
- Kernel-only Software Pipelining
- Experimental Results and Further Work

Kernel-only Software Pipelining

- Do not need to generate the Prologue and Epilogue.

Kernel-only Software Pipelining

- Do not need to generate the Prologue and Epilogue.
- This reduce “the size of the code”.
  - Reduce the pressure to binding algorithm.

Kernel-only Software Pipelining

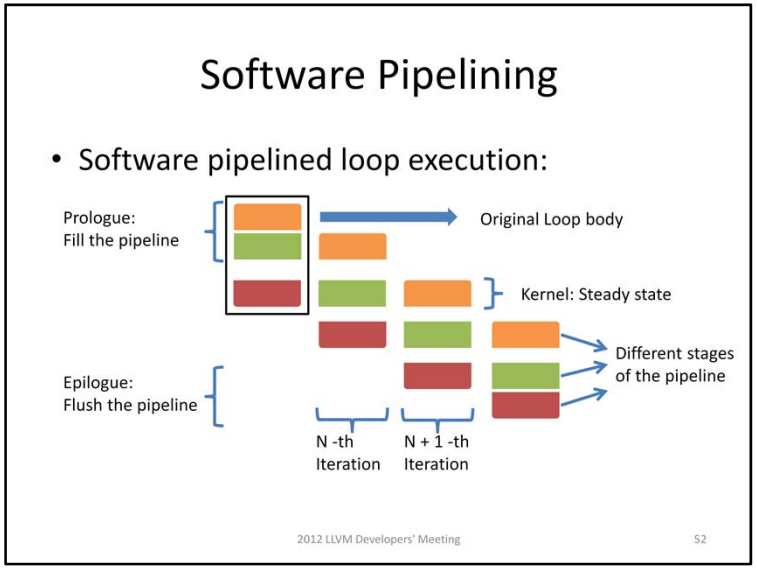
- Do not need to generate the Prologue and Epilogue.
- This reduce “the size of the code”.
  - Reduce the pressure to binding algorithm.
- Based on predicated execution.
  - All instructions are predicated in our VISA.

In fact, there are existing technique to perform kernel-only software pipelining on VLWI architecture.

Kernel-only Software Pipelining

- Do not need to generate the Prologue and Epilogue.
- This reduce “the size of the code”.
  - Reduce the pressure to binding algorithm.
- Based on predicated execution.
  - All instructions are predicated in our V-ISA.

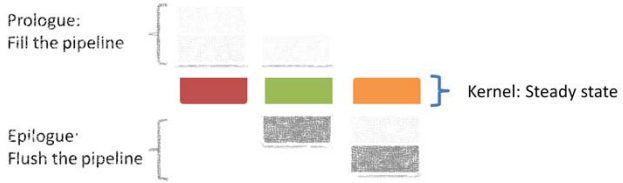
Convenience provided by CodeGen



Software pipelining is a type of out-of-order execution, except that the reordering is done by a compiler.

Software Pipelining

- The kernel is already contains all stages, why we need to duplicate them in the Prologue and Epilogue?

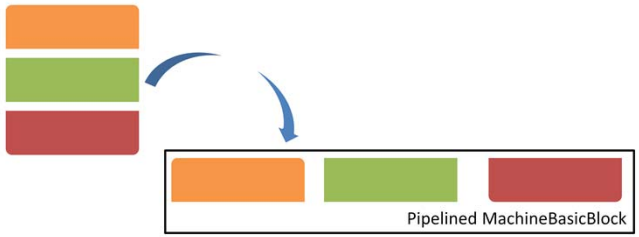


Loop Body Folding

- Loop body scheduled by Modulo Scheduling:

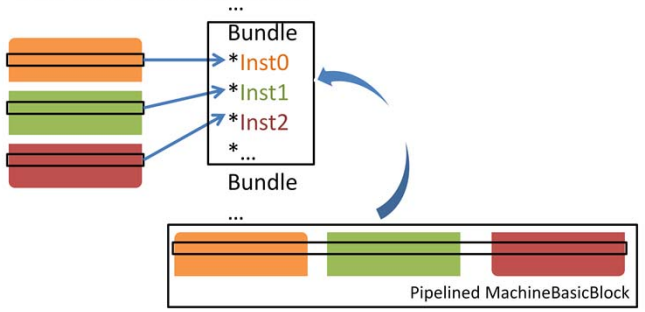
Loop Body Folding

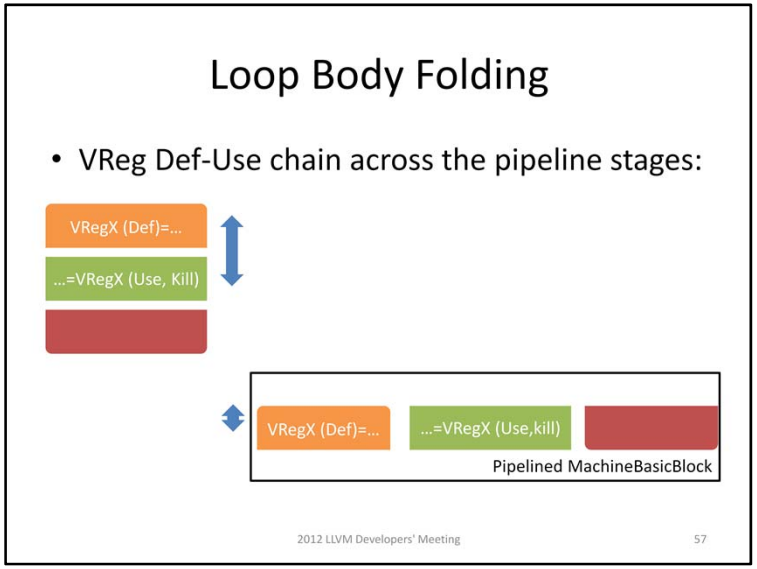
- Fold the loop body to reflect the fact that the stages are executed in parallel.



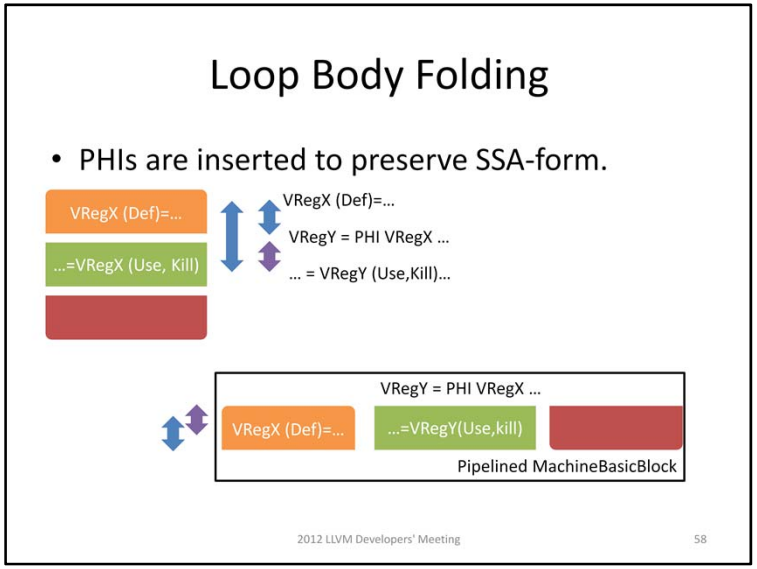
Loop Body Folding

- The Instructions in different stages are packed into the same bundle.

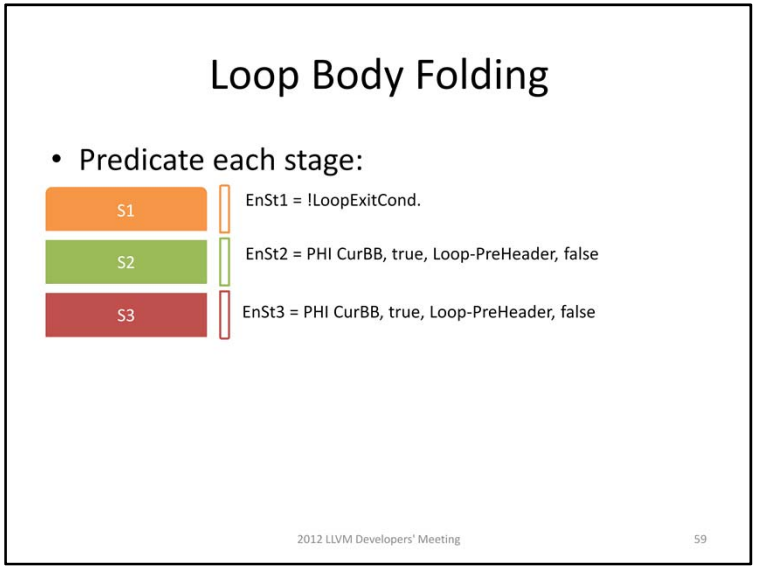




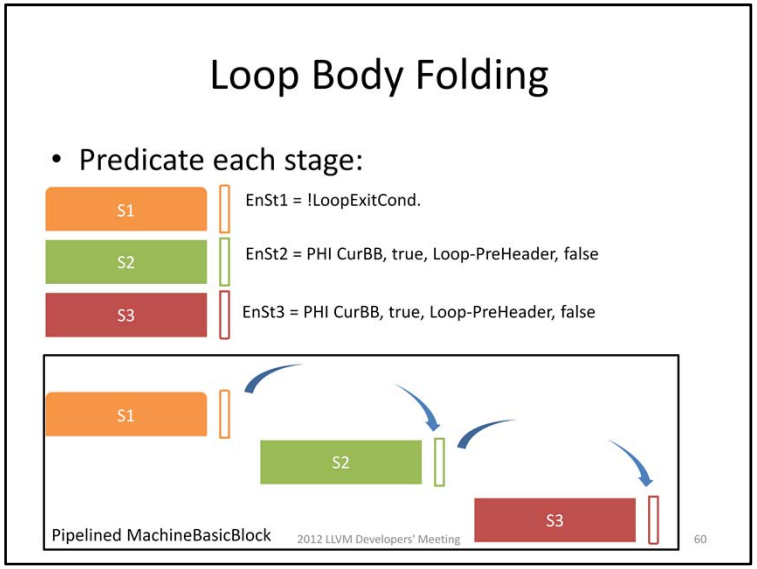
After loop-body folding the Def-use chains are also folded, this may result in a wrong live-interval and even break the SSA-form.



Hence we need to insert PHIs to break the cross-stages Def-use chain.



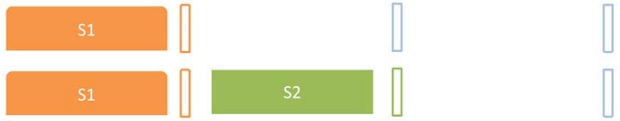
To fill and flush the pipeline correctly, we also need to predicate each stage.



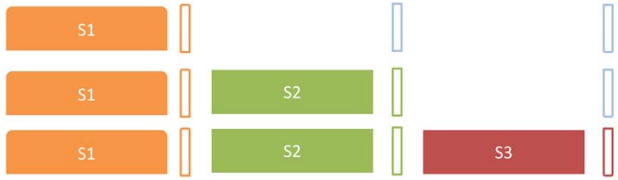
By constructing the predicate chain carefully, we can propagate the stage enable correctly.

Execution of the Folded Loop Body

Execution of the Folded Loop Body



Execution of the Folded Loop Body



...

Execution of the Folded Loop Body

Execution of the Folded Loop Body

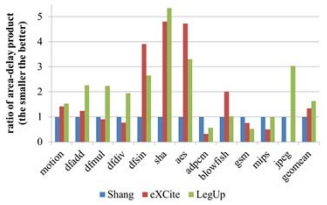
Execution of the Folded Loop Body

Outline

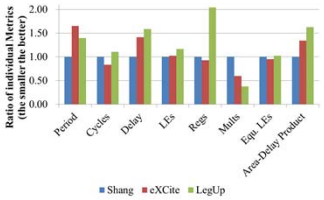
- Introduction
- Overview of the Shang HLS framework
- The Cross-level Engine
- Kernel-only Software Pipelining
- Experimental Results and Further Work

Experimental Results

For each programs in CHStone



For each metrics of the HW



Delay (us) = Period (ns) * Cycles / 1000
Equ. Les = LEs + Mults * 115
Where 115 is the LEs required to implement A 9x9 mult.

Delay is equal to the product of period and cycles, Equ. LEs is equal to LEs add the product of 115 and the Mults.

Wish List and Further work

- Access more than one MachineFunction at a time.
- Interprocedural Analyses/Optimizations with Polly.
- Compile flow for heterogeneous architecture.
- HLS-specific IR passes, e.g. do not duplicate function body when inlining.

Acknowledgements

- Thanks the people involved in this project.
  - Qingrui Liu, Junyi Le, Yuelai Yuan, …
- Thanks the LLVM community to provide the compiler infrastructure.
- Thanks SYSU to support this work.
- Thanks ADSC to pay my salary.
  - So that I can buy the flight ticket.

THANKS AND QUESTIONS?