Software Compilation Using FPGA Hardware: Register Allocation

YIMING TAN, ADITYA DIWAKAR,
JASON JAGIELO AND VINCENT MOONEY

GEORGIA INSTITUTE OF TECHNOLOGY, ATLANTA, GEORGIA, USA

presented at MECO'2023 and CPSIoT'2023, Budva, Montenegro, June 6-10
www.mecoconference.me

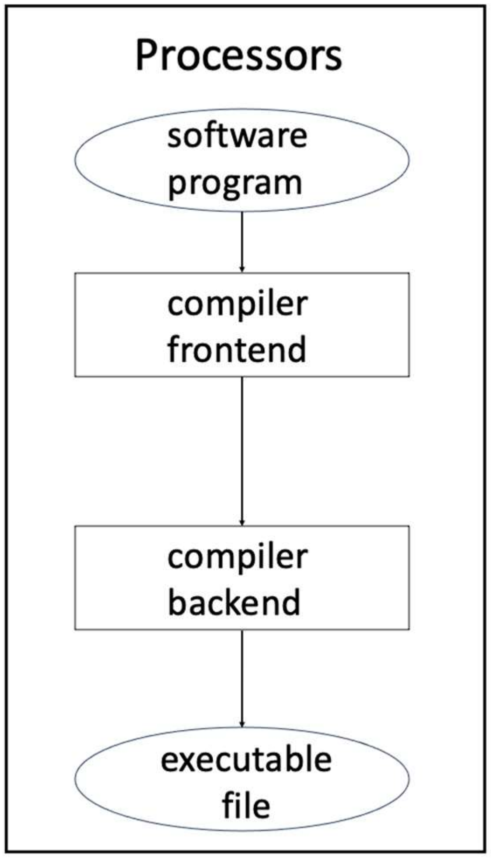
CPS&IoT

Outline

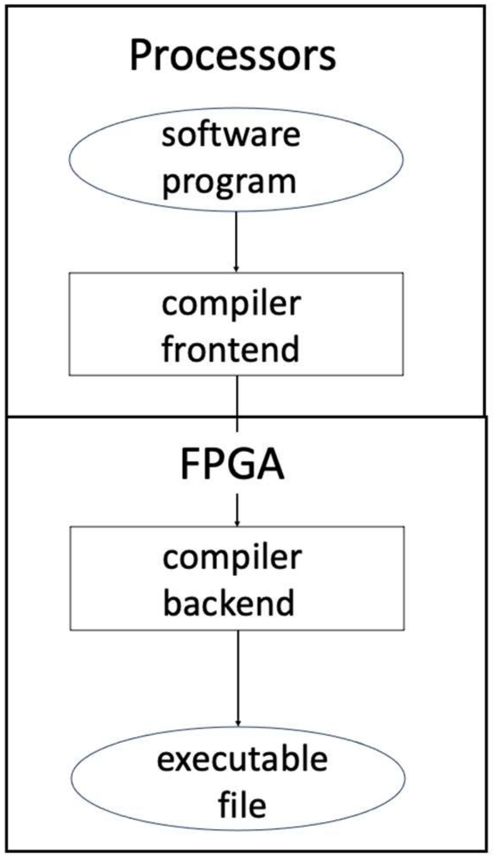
Outline

Motivation and Introduction

Traditional Approach



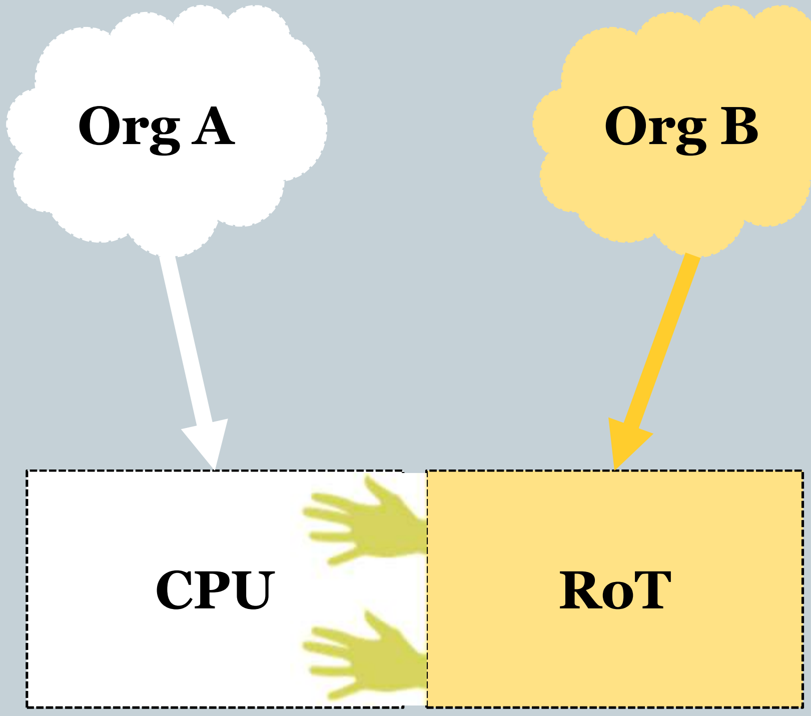
Our Approach

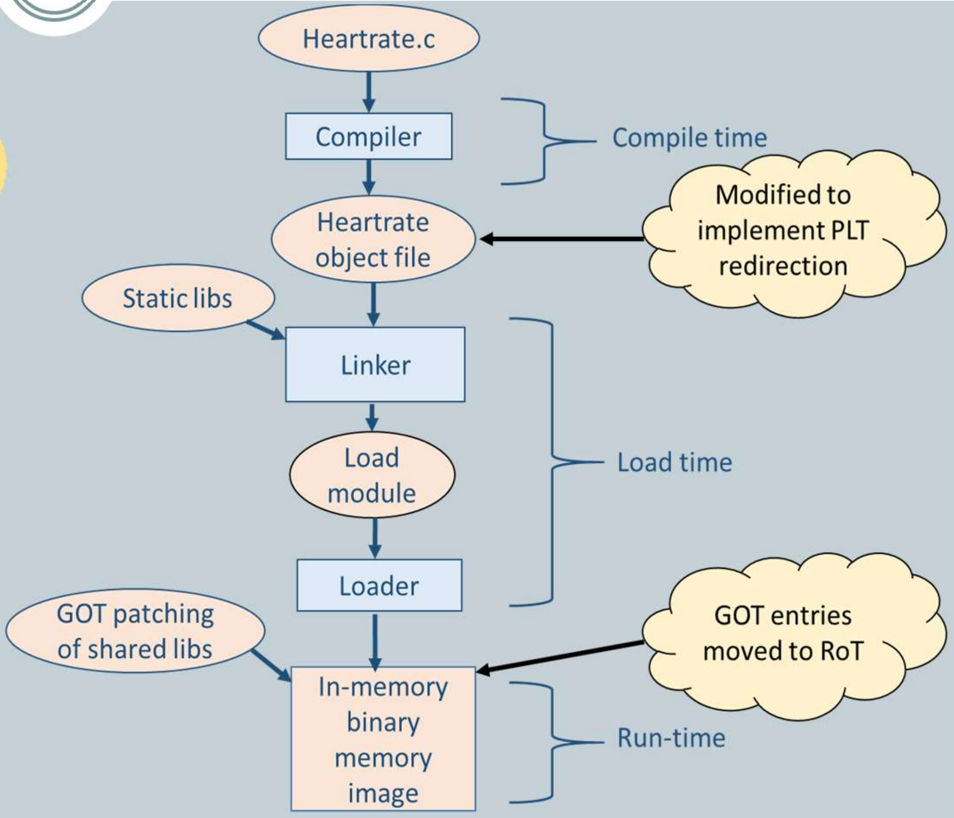


Outline

- Motivation and Introduction
- Prior Work and Background
- Methodology
- Experimental Platform and Design
- Experimental Results
- Discussion and Future Work
- Conclusions

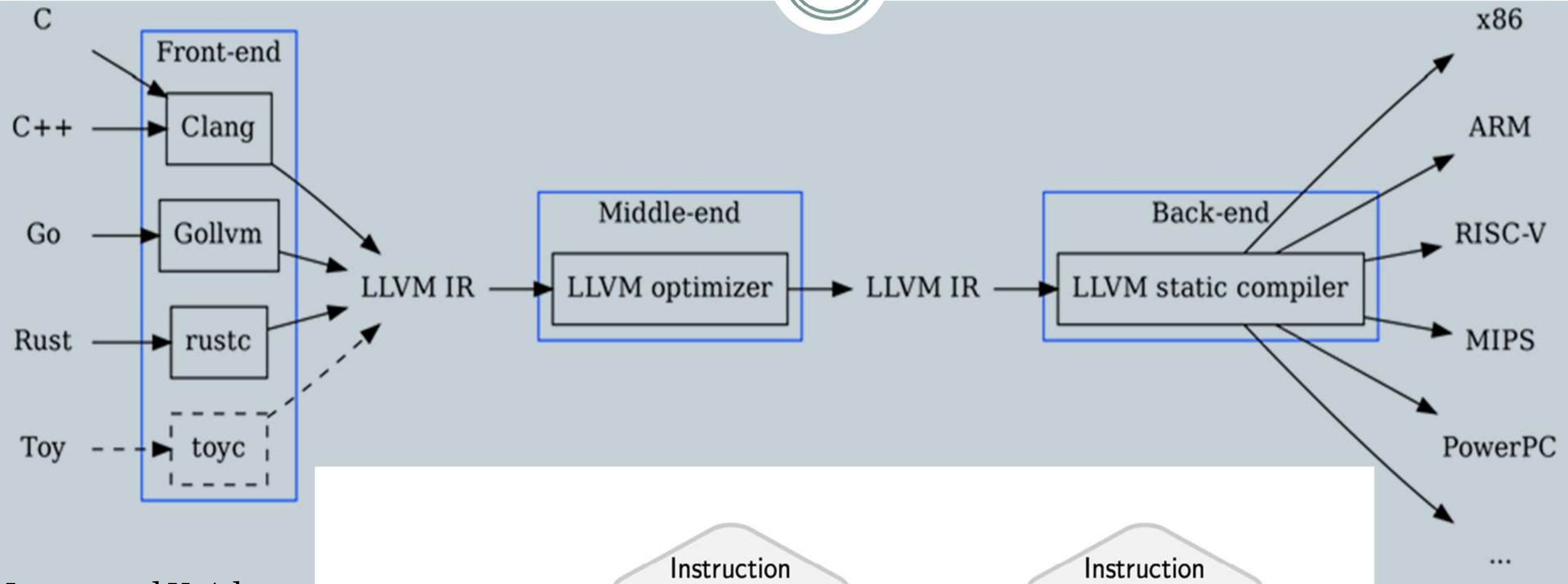
Prior Work: MECO 2019



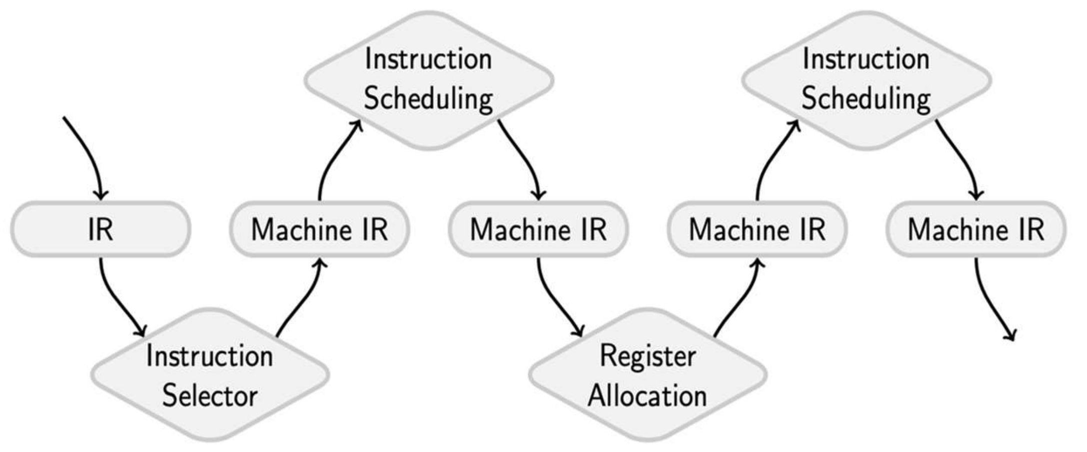


G. Lopez, M. Foreman, A. Daftardar, P. Coppock, Z. Tolaymat, and V. J. Mooney, "Hardware root-of-trust based integrity for shared library function pointers in embedded systems," in *8th Mediterranean Conference on Embedded Computing (MECO)*, 2019, pp. 1–6.

Background: Software Compilation



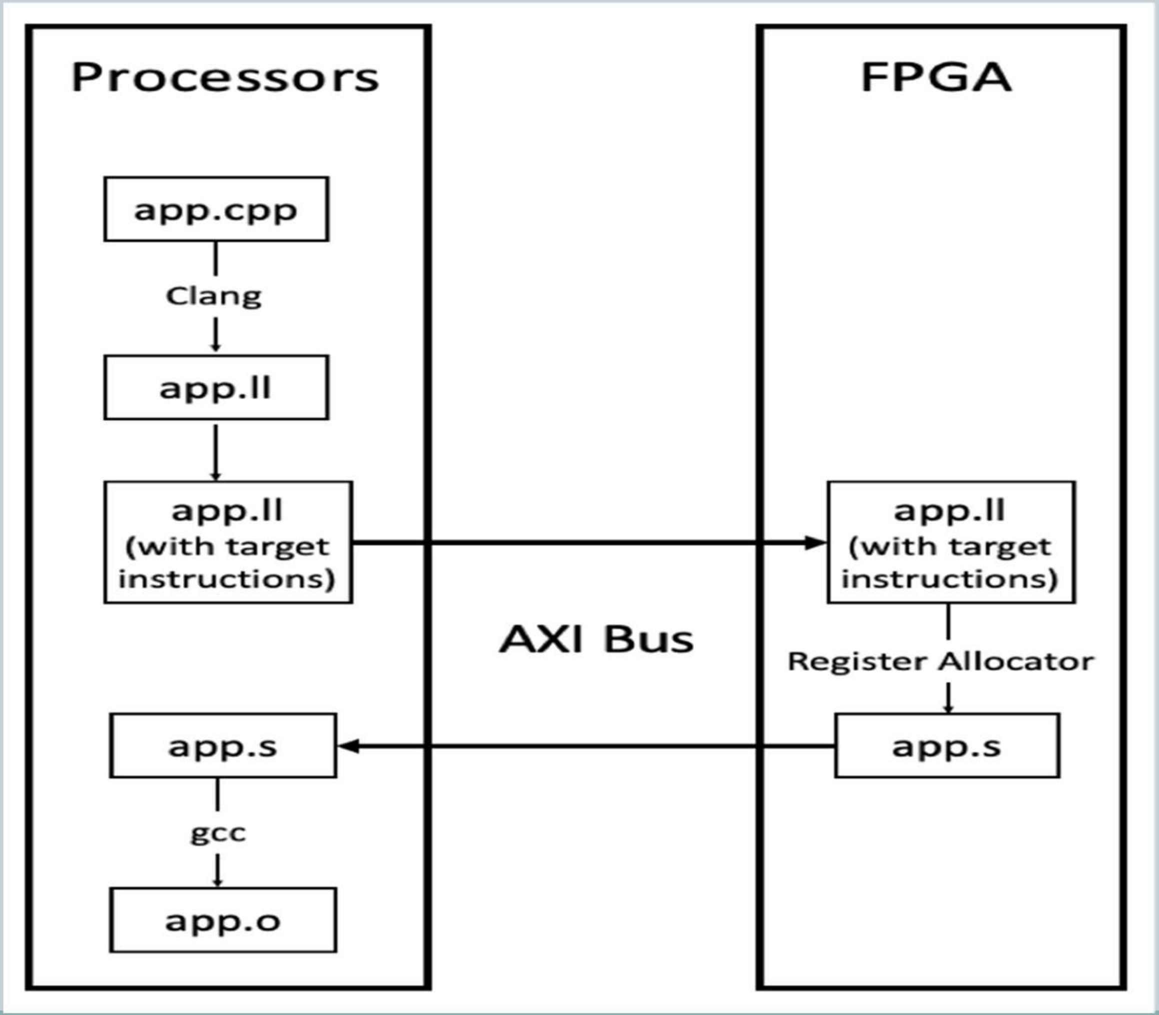
C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in Proceedings of International Symposium on Code Generation and Optimization, March 2004, pp. 75–86.



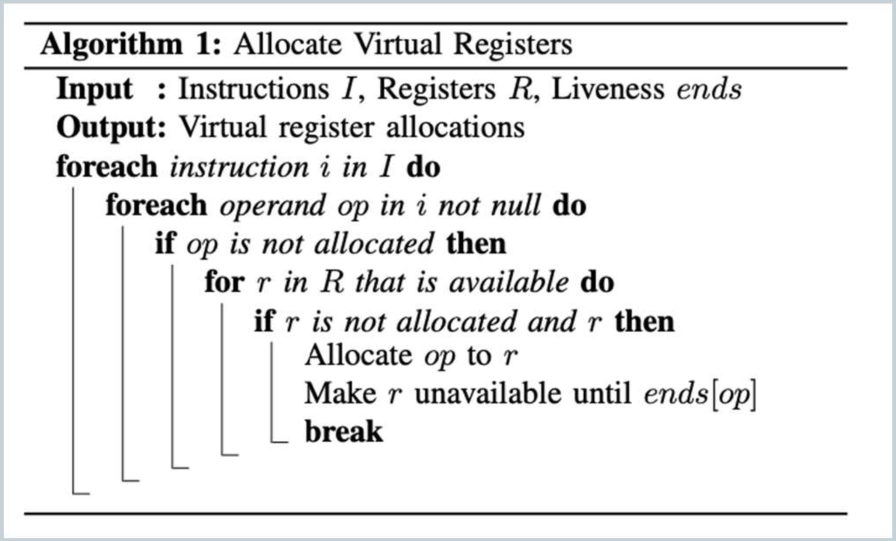
A. Krall "Compiler Backend Generation from Structural Processor Models," *Technische Universitat Wien*, Oct, 2009. *https://perso.telecom-paristech.fr/brandner/paper/thesis_brandner_2009.pdf* Last accessed: Nov 17, 2021.

Outline

Methodology: Design Flow



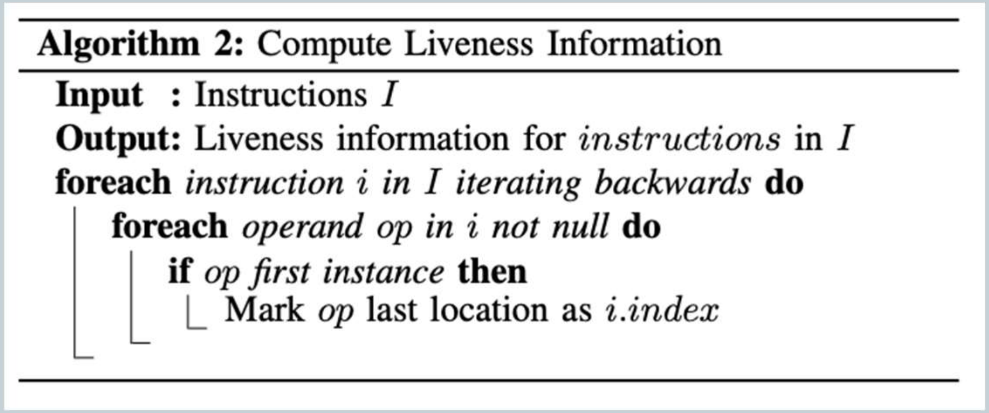
Methodology: Register Allocation Algorithms



```
Algorithm 1: Allocate Virtual Registers
Input  : Instructions $I$, Registers $R$, Liveness $ends$
Output: Virtual register allocations
foreach instruction $i$ in $I$ do
    foreach operand $op$ in $i$ not null do
        if $op$ is not allocated then
            for $r$ in $R$ that is available do
                if $r$ is not allocated and $r$ then
                    Allocate $op$ to $r$
                    Make $r$ unavailable until $ends[op]$
                    break
```

Algorithm 1 implements a register allocation algorithm based on the liveness information from Algorithm 2.

Algorithm 2 implements register liveness calculation using known techniques.



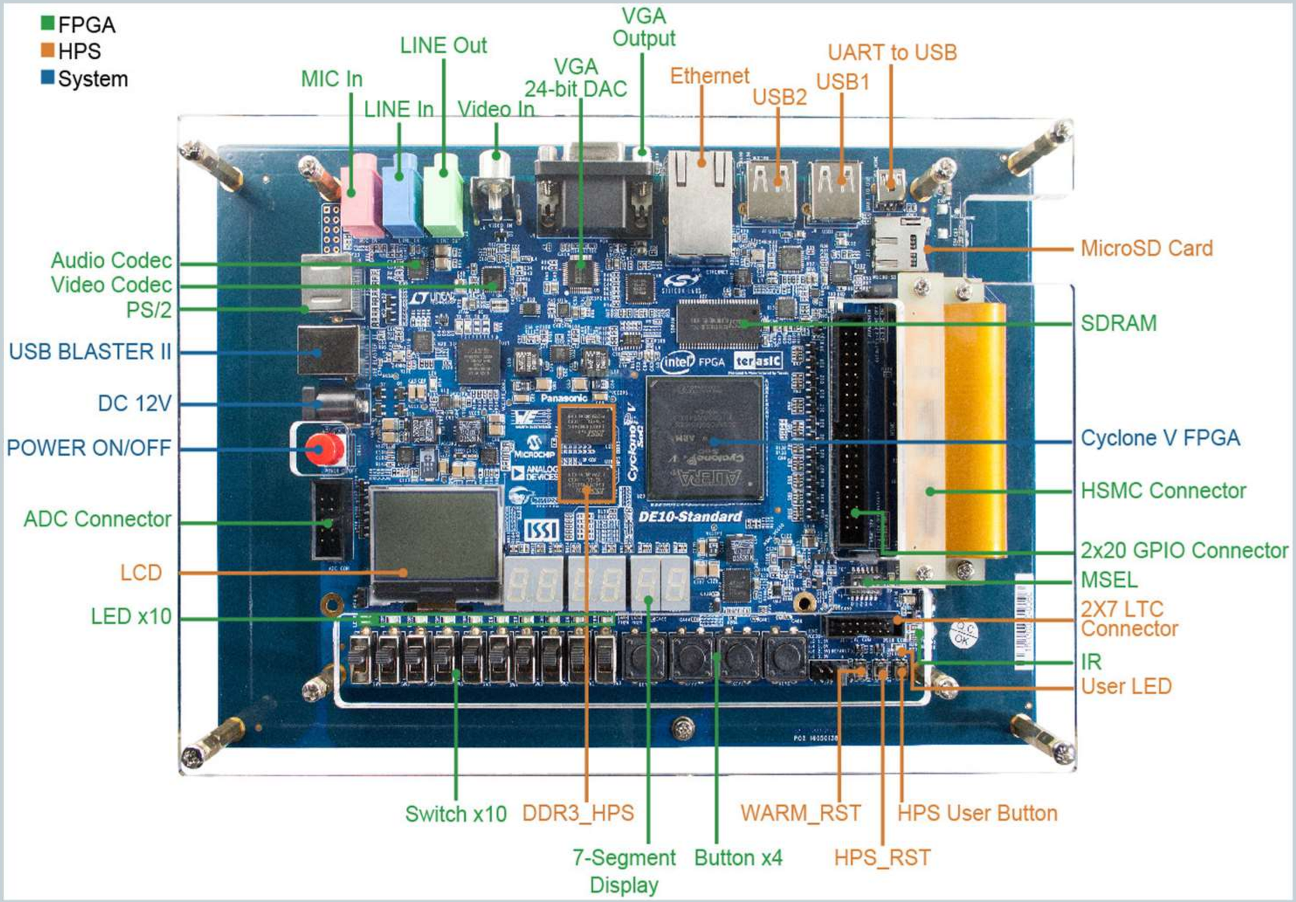
```
Algorithm 2: Compute Liveness Information
Input  : Instructions $I$
Output: Liveness information for $instructions$ in $I$
foreach instruction $i$ in $I$ iterating backwards do
    foreach operand $op$ in $i$ not null do
        if $op$ first instance then
            Mark $op$ last location as $i.index$
```

Outline

Experimental Platform

Intel DE-10 Standard Board

- ARM-based HPS
- Cyclone V FPGA



Terasic, "DE-10 standard user manual," 2017, last accessed 29 Apr 2023. [Online]. Available: https://ftp.intel.com/Public/Pub/fpgaup/pub/Intel_Material/Boards/DE10-Standard/DE10_Standard_User_Manual.pdf

Design Specifics Part 1

- Data Transfer: AXI bus
- Register Liveness
  - Three enumerated data types
  - Eleven states finite state machine
  - Takes in gcd instructions
  - Extracts opcode and registers
  - Iterates through each gcd instruction to find where each virtual register ends
  - Outputs live range for each virtual register

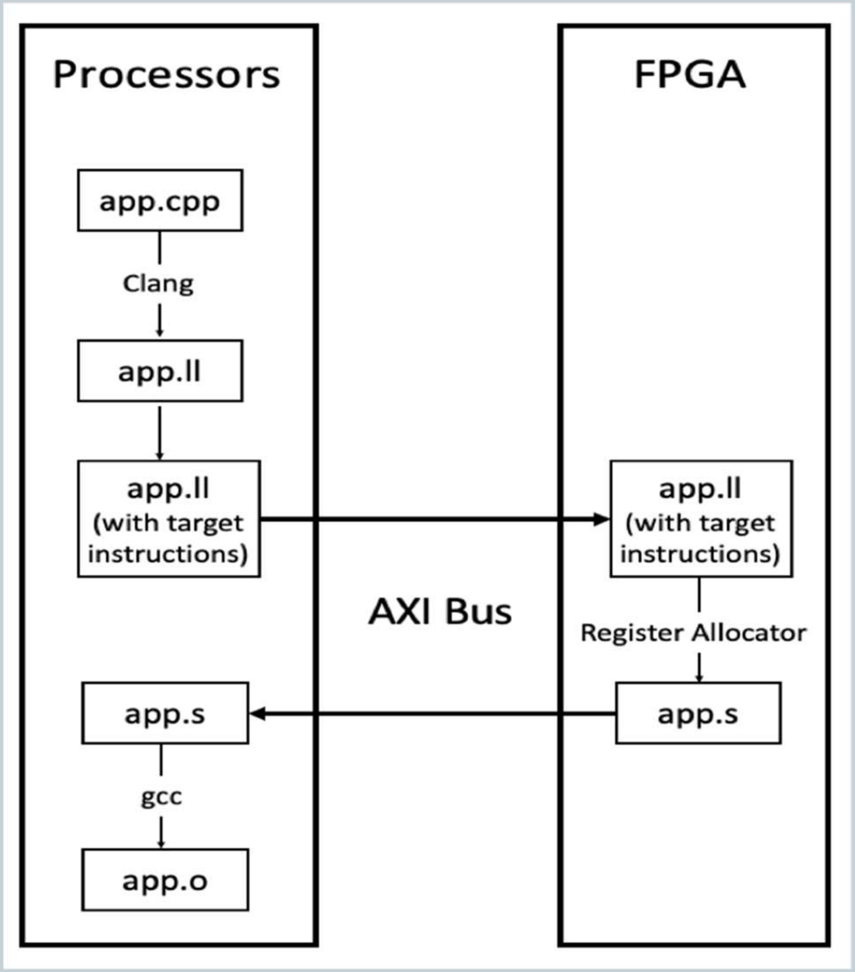
Design Specifics Part 2

- Register Allocation
  - Has four extra crucial states to find free physical registers
    - S1: whether or not the current virtual register has been mapped to a physical register
    - S2: whether any physical register is available
    - S3: which physical register is available next in line
    - S4: recording the mapping between the virtual register and the physical register
  - Deals with memory spills
  - Frees the physical registers that were allocated to virtual registers that are no longer live
  - Outputs a mapping between all virtual registers used in gcd instructions and their corresponding physical registers

Outline

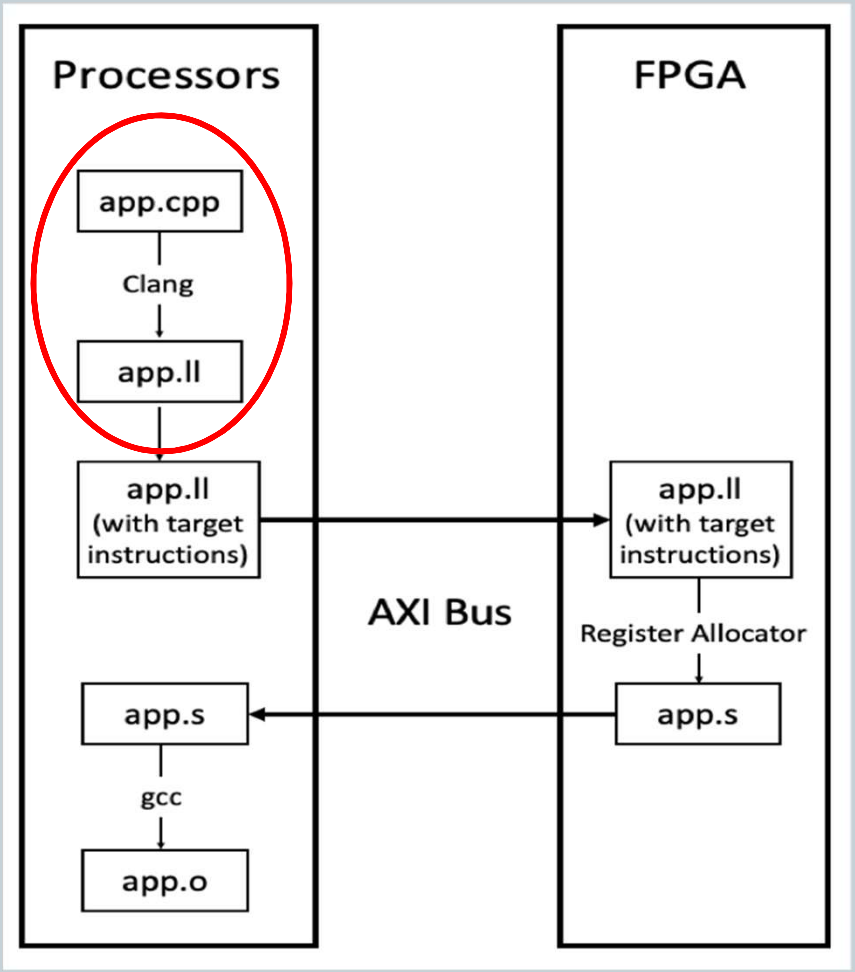
Experiment

1. Use Clang to emit app.ll from app.cpp on a laptop (note we used gcd.c for app.cpp)
2. Emit instructions in binary-IR format from app.ll on the laptop to the HPS
3. Transfer instructions in binary-IR format from the HPS to the FPGA interfacing through the AXI bus
4. Place instructions in binary-IR format in the FPGA via hardcoding in the System Verilog files for register allocation
5. Register allocation algorithms in System Verilog which read the binary-IR instructions and perform physical register replacement on the FPGA
6. Generate the assembly file app.s from instructions with physical registers on the FPGA
7. Manually read app.s from the FPGA
8. Use gcc to generate app.o on the laptop
9. Link the executable app.o on the laptop



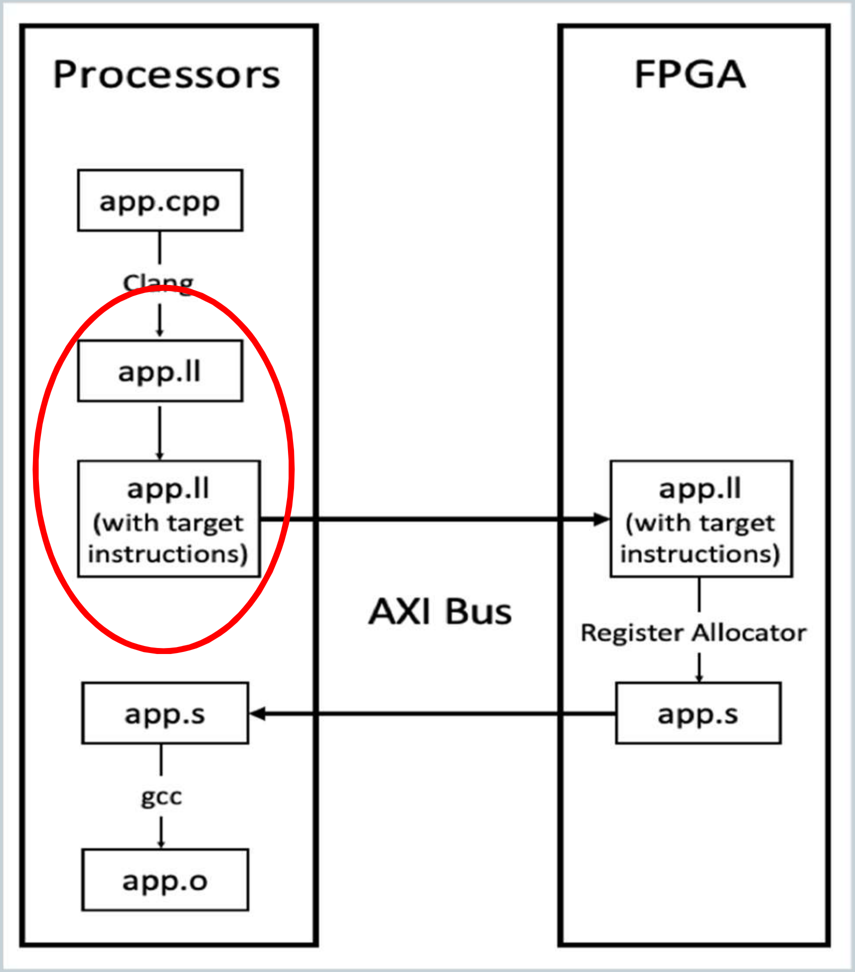
Experiment

1. Use Clang to emit app.ll from app.cpp on a laptop (note we used gcd.c for app.cpp)



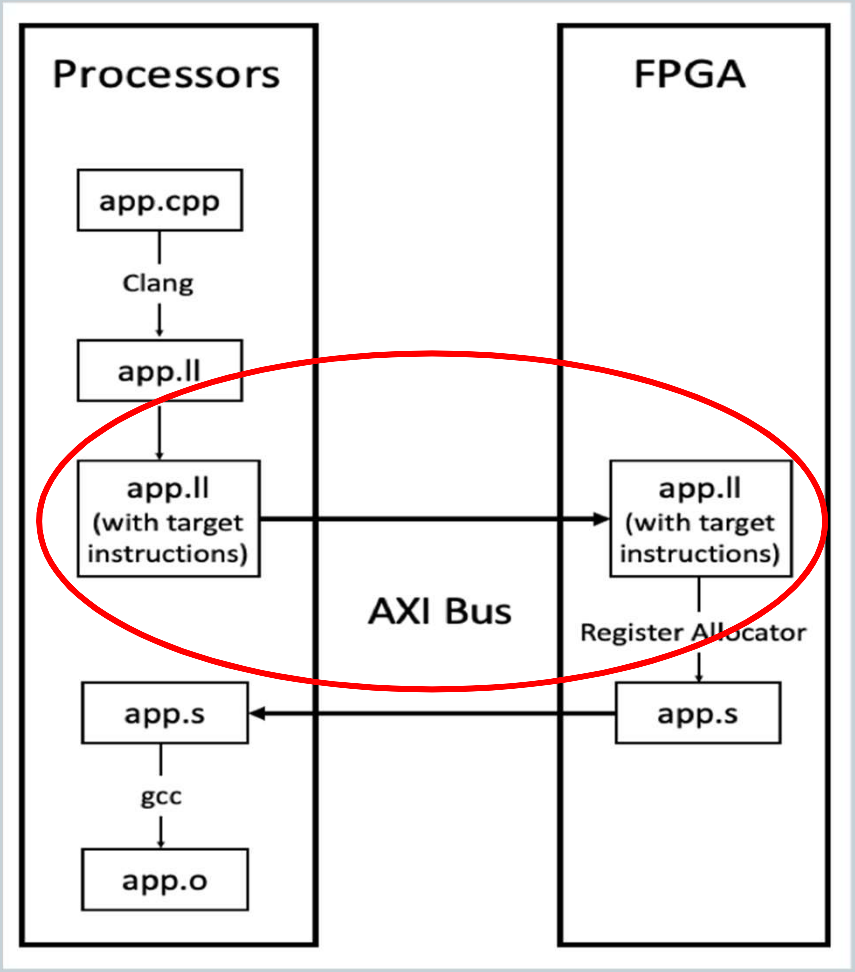
Experiment

1. Use Clang to emit app.ll from app.cpp on a laptop (note we used gcd.c for app.cpp)
2. Emit instructions in binary-IR format from app.ll on the laptop to the HPS
3.



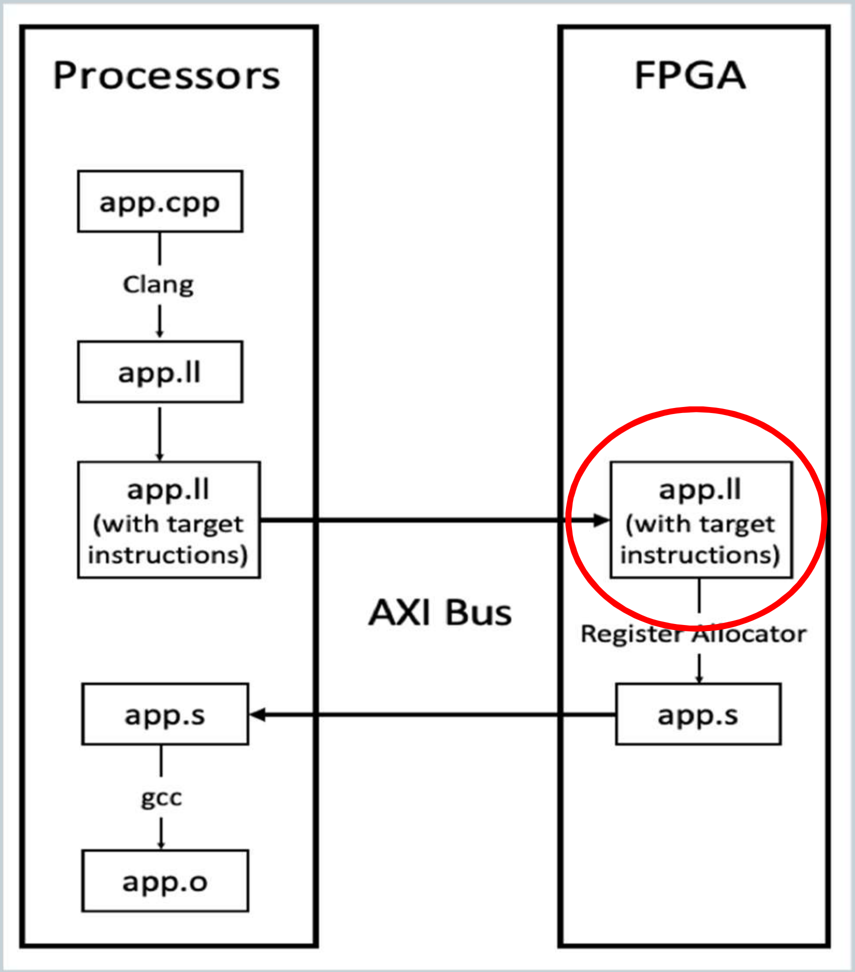
Experiment

1. Use Clang to emit app.ll from app.cpp on a laptop (note we used gcd.c for app.cpp)
2. Emit instructions in binary-IR format from app.ll on the laptop to the HPS
3. Transfer instructions in binary-IR format from the HPS to the FPGA interfacing through the AXI bus
4.



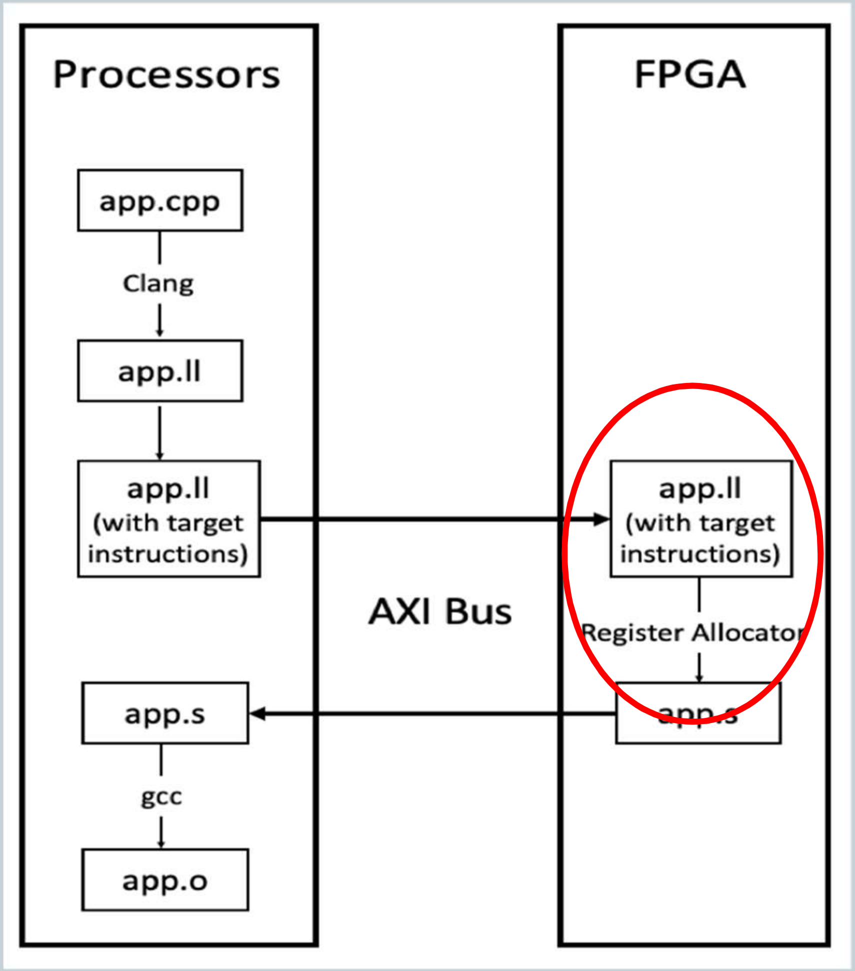
Experiment

1. Use Clang to emit app.ll from app.cpp on a laptop (note we used gcd.c for app.cpp)
2. Emit instructions in binary-IR format from app.ll on the laptop to the HPS
3. Transfer instructions in binary-IR format from the HPS to the FPGA interfacing through the AXI bus
4. Place instructions in binary-IR format in the FPGA via hardcoding in the System Verilog files for register allocation
5.



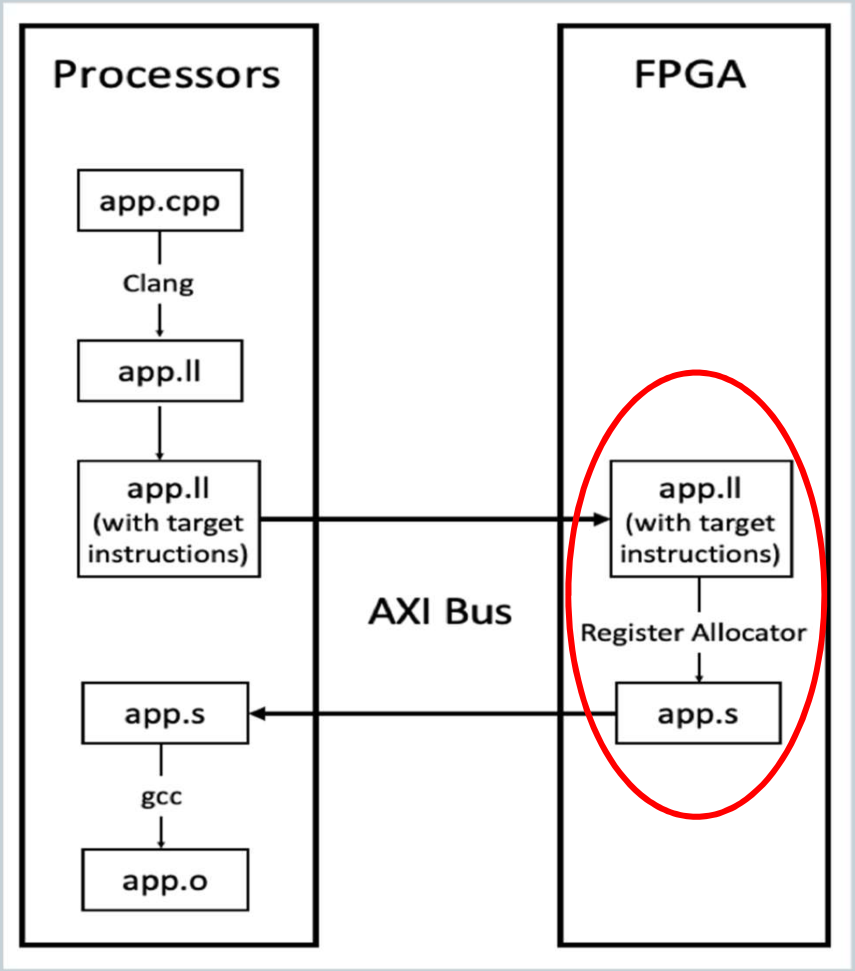
Experiment

1. Use Clang to emit app.ll from app.cpp on a laptop (note we used gcd.c for app.cpp)
2. Emit instructions in binary-IR format from app.ll on the laptop to the HPS
3. Transfer instructions in binary-IR format from the HPS to the FPGA interfacing through the AXI bus
4. Place instructions in binary-IR format in the FPGA via hardcoding in the System Verilog files for register allocation
5. Register allocation algorithms in System Verilog which read the binary-IR instructions and perform physical register replacement on the FPGA
6.



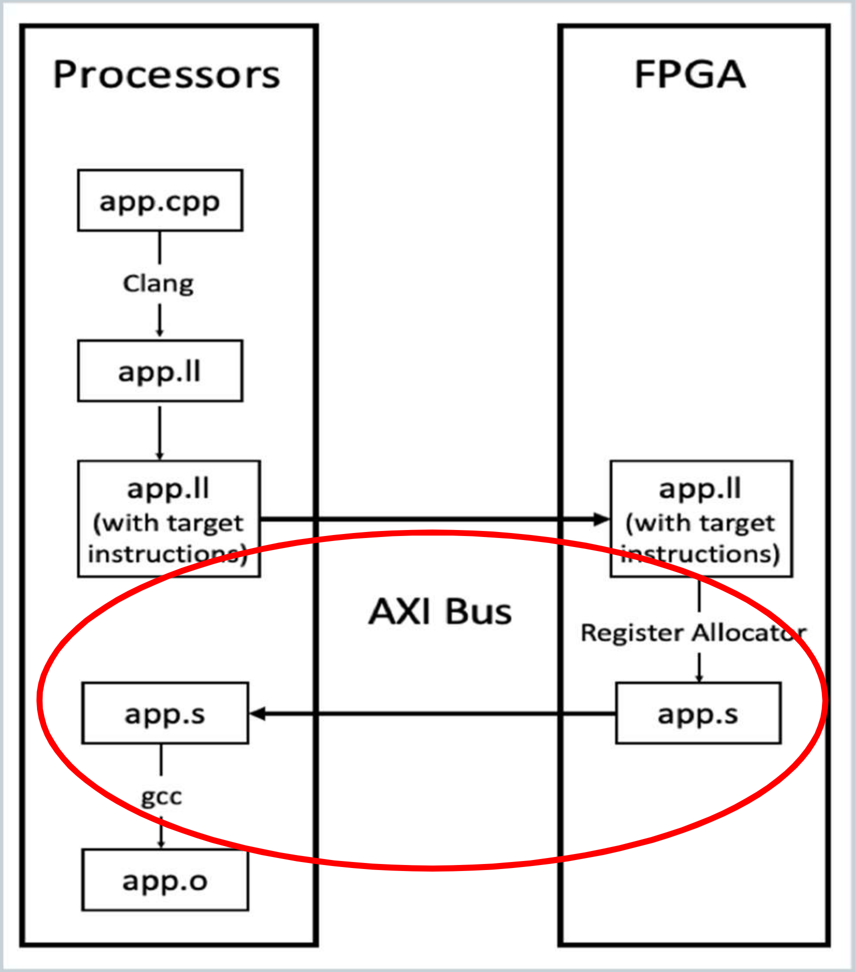
Experiment

1. Use Clang to emit app.ll from app.cpp on a laptop (note we used gcd.c for app.cpp)
2. Emit instructions in binary-IR format from app.ll on the laptop to the HPS
3. Transfer instructions in binary-IR format from the HPS to the FPGA interfacing through the AXI bus
4. Place instructions in binary-IR format in the FPGA via hardcoding in the System Verilog files for register allocation
5. Register allocation algorithms in System Verilog which read the binary-IR instructions and perform physical register replacement on the FPGA
6. Generate the assembly file app.s from instructions with physical registers on the FPGA
7.
8.
9.



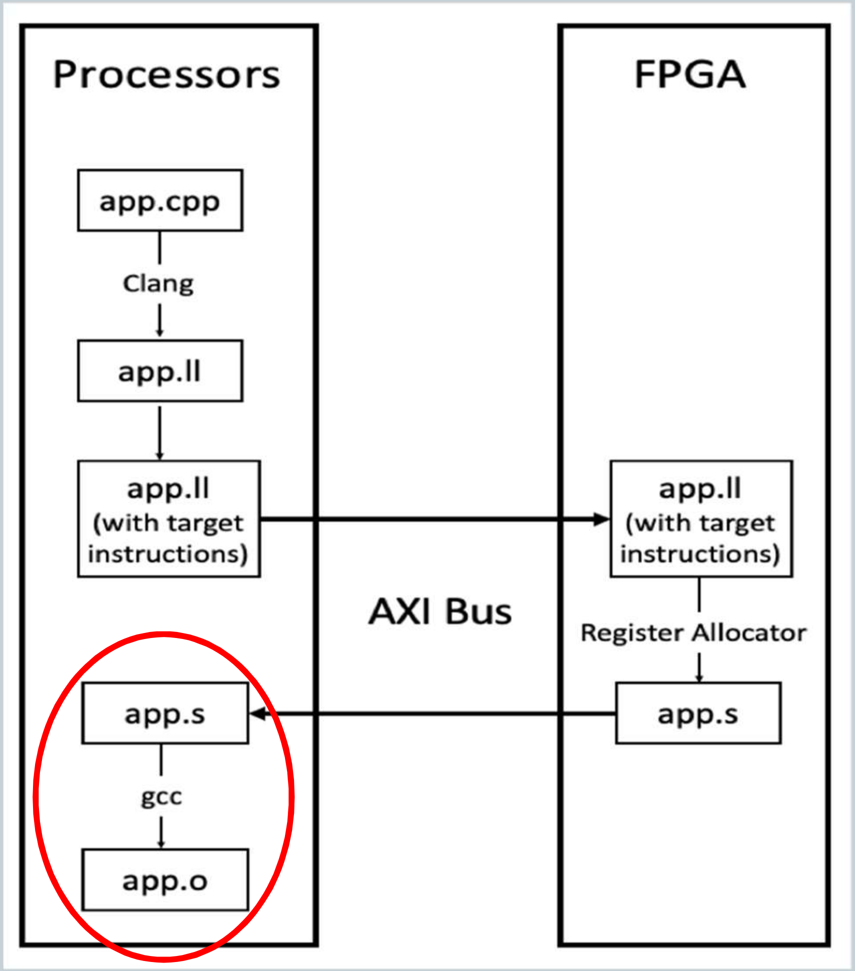
Experiment

1. Use Clang to emit app.ll from app.cpp on a laptop (note we used gcd.c for app.cpp)
2. Emit instructions in binary-IR format from app.ll on the laptop to the HPS
3. Transfer instructions in binary-IR format from the HPS to the FPGA interfacing through the AXI bus
4. Place instructions in binary-IR format in the FPGA via hardcoding in the System Verilog files for register allocation
5. Register allocation algorithms in System Verilog which read the binary-IR instructions and perform physical register replacement on the FPGA
6. Generate the assembly file app.s from instructions with physical registers on the FPGA
7. Manually read app.s from the FPGA
8.
9.



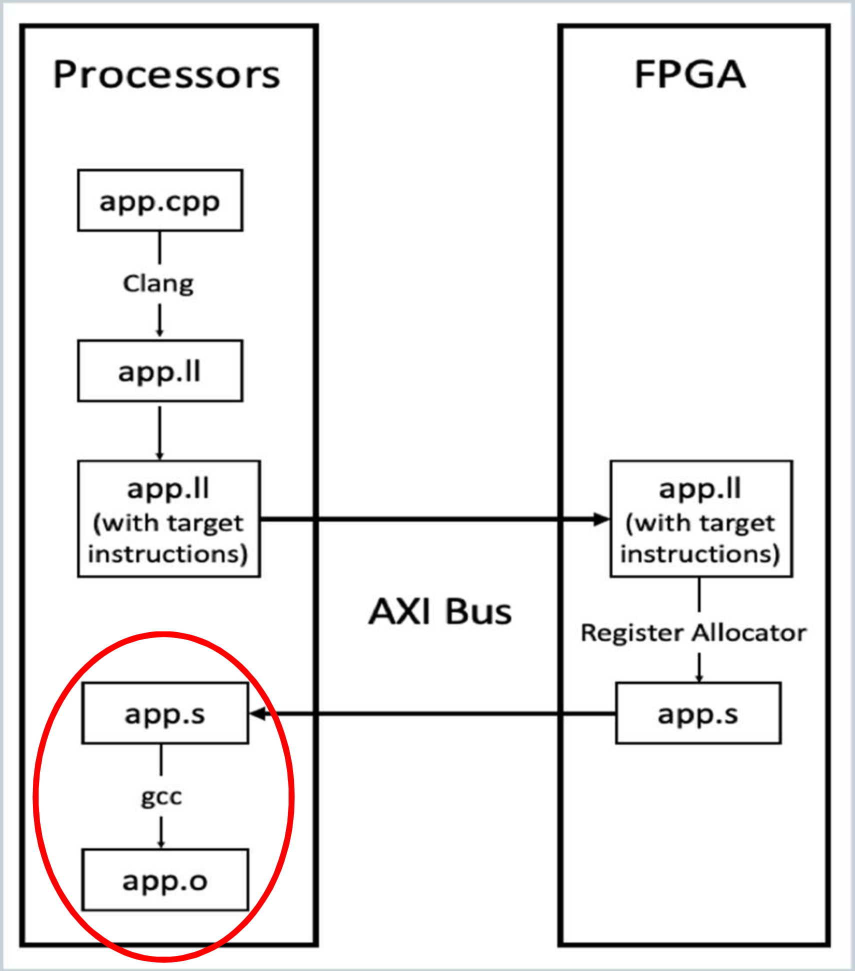
Experiment

1. Use Clang to emit app.ll from app.cpp on a laptop (note we used gcd.c for app.cpp)
2. Emit instructions in binary-IR format from app.ll on the laptop to the HPS
3. Transfer instructions in binary-IR format from the HPS to the FPGA interfacing through the AXI bus
4. Place instructions in binary-IR format in the FPGA via hardcoding in the System Verilog files for register allocation
5. Register allocation algorithms in System Verilog which read the binary-IR instructions and perform physical register replacement on the FPGA
6. Generate the assembly file app.s from instructions with physical registers on the FPGA
7. Manually read app.s from the FPGA
8. Use gcc to generate app.o on the laptop
9.



Experiment

1. Use Clang to emit app.ll from app.cpp on a laptop (note we used gcd.c for app.cpp)
2. Emit instructions in binary-IR format from app.ll on the laptop to the HPS
3. Transfer instructions in binary-IR format from the HPS to the FPGA interfacing through the AXI bus
4. Place instructions in binary-IR format in the FPGA via hardcoding in the System Verilog files for register allocation
5. Register allocation algorithms in System Verilog which read the binary-IR instructions and perform physical register replacement on the FPGA
6. Generate the assembly file app.s from instructions with physical registers on the FPGA
7. Manually read app.s from the FPGA
8. Use gcc to generate app.o on the laptop
9. Link the executable app.o on the laptop



Experimental Result Tables I and II

Table I shows the resource utilization on the FPGA is efficient.

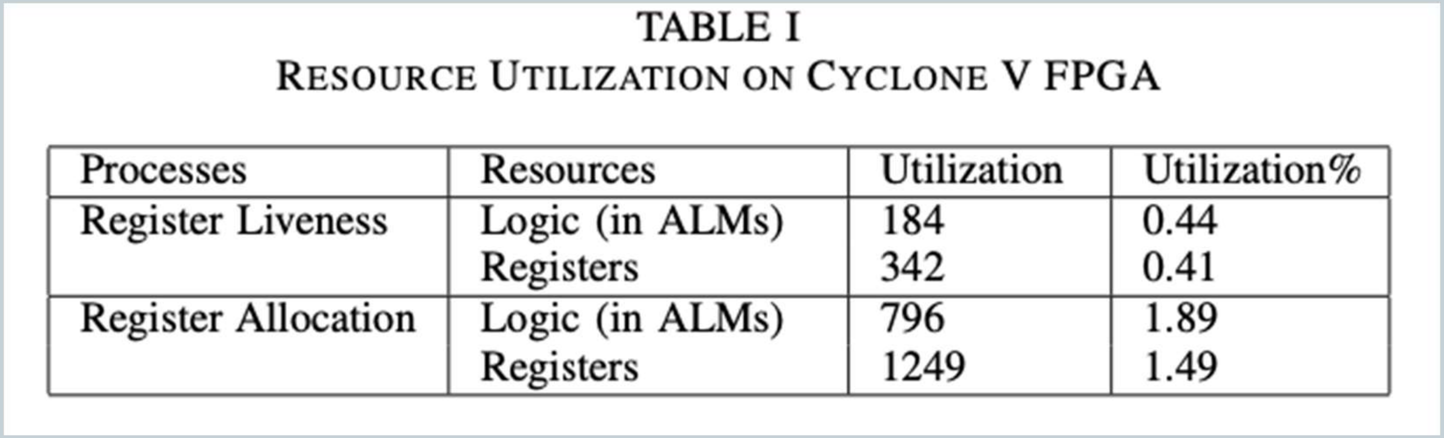


TABLE I
RESOURCE UTILIZATION ON CYCLONE V FPGA

Processes	Resources	Utilization	Utilization%
Register Liveness	Logic (in ALMs)	184	0.44
	Registers	342	0.41
Register Allocation	Logic (in ALMs)	796	1.89
	Registers	1249	1.49

Table II shows the time Quartus takes to generate FPGA bitstreams.

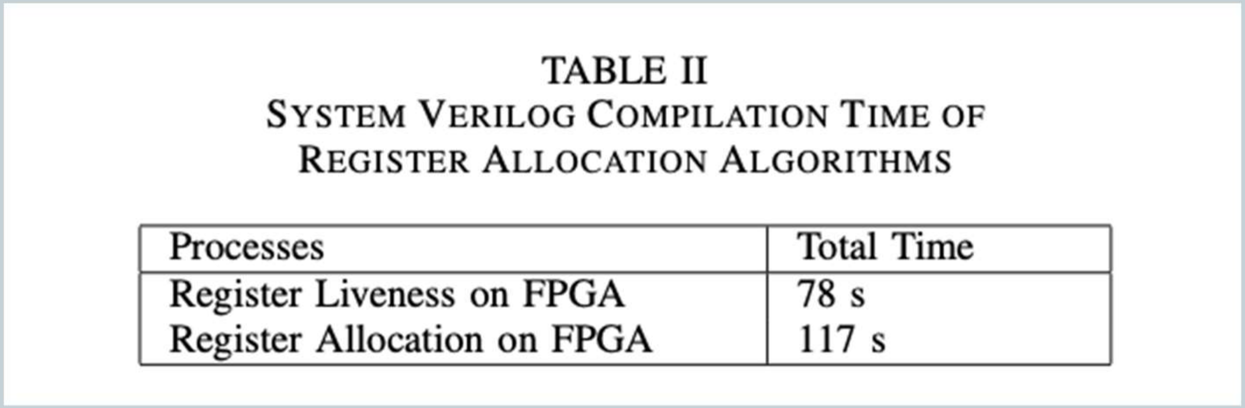


TABLE II
SYSTEM VERILOG COMPILATION TIME OF REGISTER ALLOCATION ALGORITHMS

Processes	Total Time
Register Liveness on FPGA	78 s
Register Allocation on FPGA	117 s

Experimental Result Tables III and IV

Table III shows the average execution time at 50 MHz of gcd over different inputs.

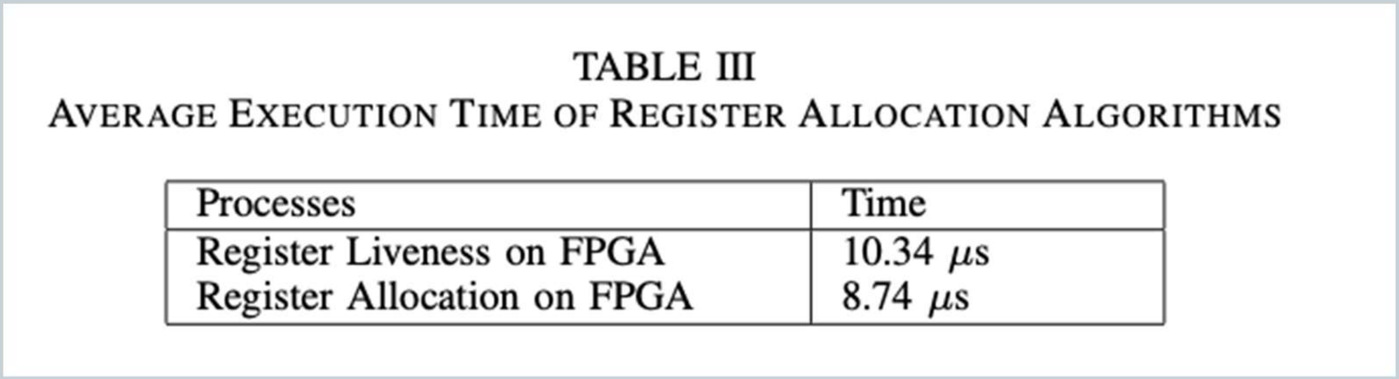


TABLE III
AVERAGE EXECUTION TIME OF REGISTER ALLOCATION ALGORITHMS

Processes	Time
Register Liveness on FPGA	10.34 μs
Register Allocation on FPGA	8.74 μs

Table III does not include AXI bus communication time. Table IV is needed to estimate HPS to FPGA and FPGA to HPS communication overhead.

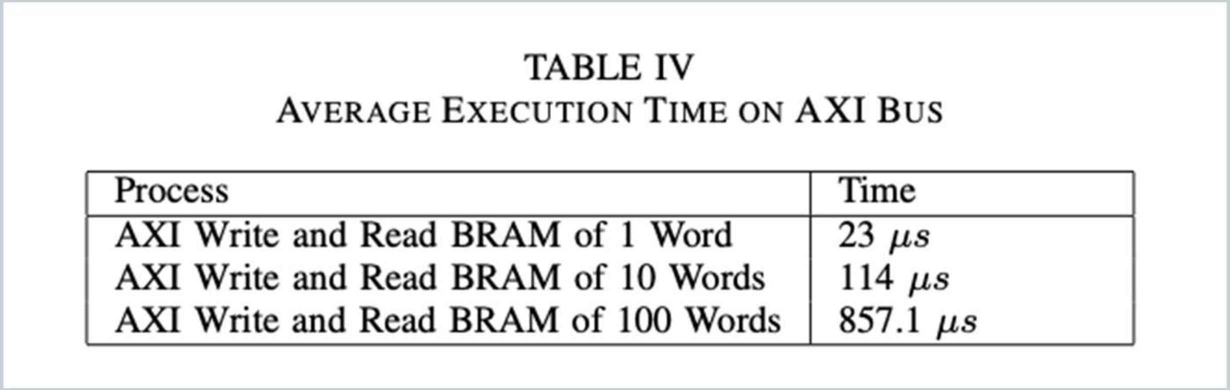


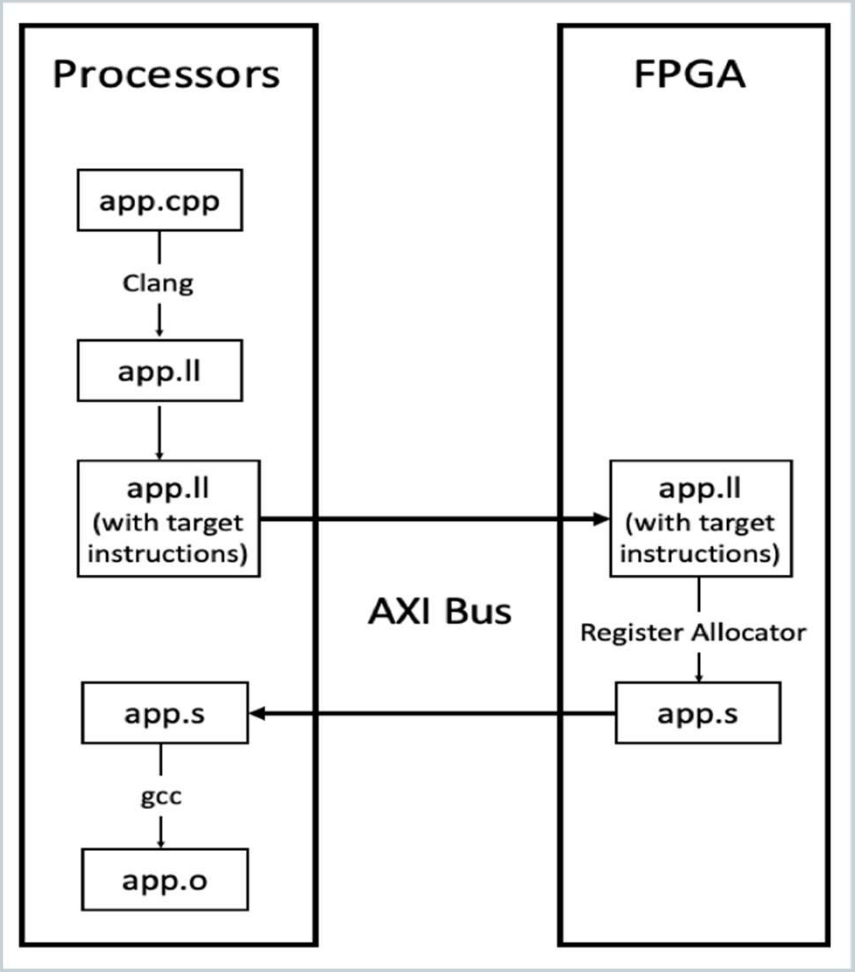
TABLE IV
AVERAGE EXECUTION TIME ON AXI BUS

Process	Time
AXI Write and Read BRAM of 1 Word	23 μs
AXI Write and Read BRAM of 10 Words	114 μs
AXI Write and Read BRAM of 100 Words	857.1 μs

Outline

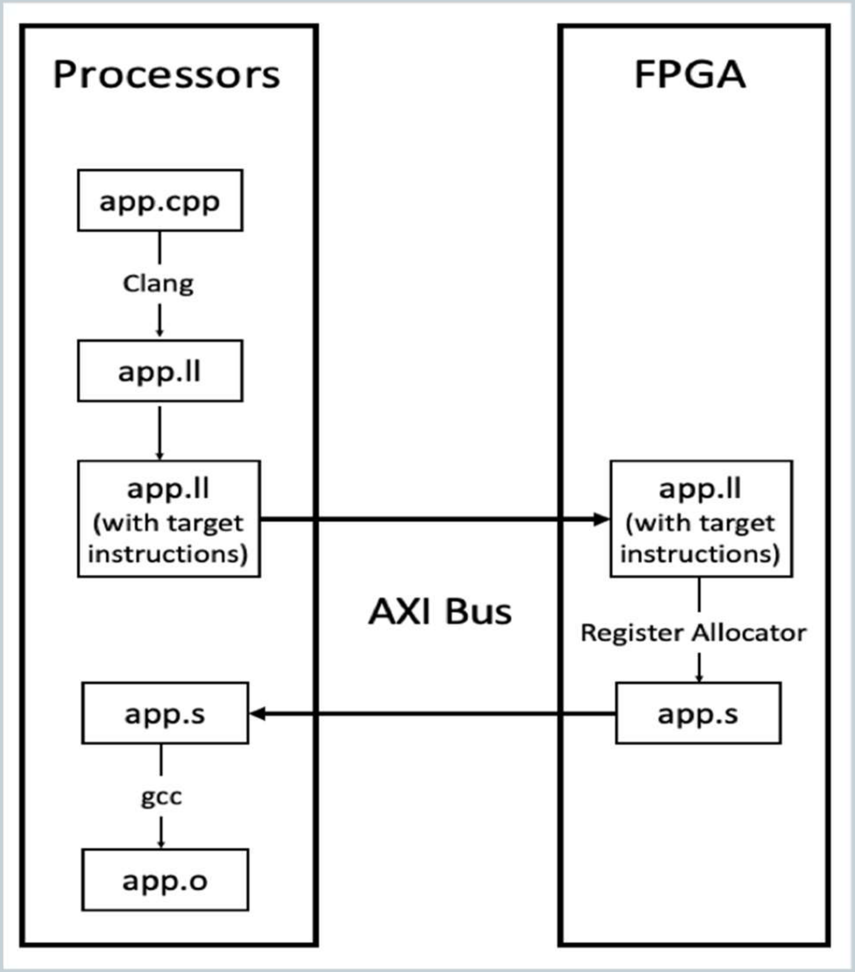
Discussion

- FPGA register allocation fully functional with gcd hardcoded in System Verilog
  - No aspect of our methodology is specific to any particular FPGA platform
- Data transfer not integrated
- LLVM software compilation flow fully functional including final generation of executable assembly code (gcd.o)



Future Work in Red

1. Use Clang to emit app.ll from app.cpp on the HPS of the board
2. Emit instructions in binary-IR format from app.ll on the HPS
3. Transfer instructions in binary-IR format from the HPS to the FPGA BRAM interfacing through the AXI bus
4. Register allocation algorithms in System Verilog which read the binary-IR instructions and perform physical register replacement on the FPGA
5. Generate the assembly file app.s from instructions with physical registers on the FPGA
6. Send app.s from the FPGA to the HPS
7. Use gcc to generate app.o on the HPS
8. Link then execute app.o on the HPS



Outline

Conclusions

- Have shown a proof-of-concept demonstration of how to compile software on an FPGA which cannot be attacked at run-time (i.e., the FPGA used is not dynamically re-configurable).
- Implemented the backend register allocation step in System Verilog and have compiled a gcd program to ARM assembly.
  - Frontend software compilation in FPGA is left as future work
- This research aims to enable full just-in-time compilation on an FPGA at run-time which is protected from cyberattack by implementation in hardware.

THANK YOU

Q&A