International Journal of Applied Engineering Research
ISSN 0973-4562 Volume 10, Number 12 (2015) pp. 30699-30712
© Research India Publications
http://www.ripublication.com

Resource Estimation of Programs for Reconfigurable Computing Systems

Ashish Mishra[1], Vivek Vanga[2], Vani V.[3], Abhijit Rameshwar Asati[4], Kota Solomon Raju[5]

[1, 2,3,4] *Department of Electrical and Electronics Engineering,*
BITS-Pilani, Pilani, Rajasthan, India,
{ashishmishra[1], f2009128[2],h2013126[3]abhijit_asati[4]},@pilani.bits-pilani.ac.in,
Digital System Group, CSIR-CEERI, Pilani, Rajasthan, India,
solomon@ceeri.ernet.in[5]

Abstract

Many practical applications developed in high level languages(HLL) like C, C++ when synthesized into a hardware description (HDL), using the high level synthesis tool, span large amount of space on the chip. In reconfigurable computing systems (RCS) when such a design is interfaced and mapped with the processor bus following a system-on-chip flow, the total area required may fall short due to a number of other peripherals present in the design. For mapping such an application, the solution is migrating part of the design to hardware (HW) and the remaining part to software (SW). Instead of following HW-SW approach, in this paper, we propose a novel design flow to map such an application using the concept of partial reconfiguration (PR) on field programmable gate arrays (FPGA). The design flow proposes a robust method to partition the application into n parts, determined by giving a maximum possible area on the chip. The process of mapping logic from HLL to HDL requires resource estimation at the granularity level of instruction. Hence, using a compiler the control flow graphs (CFG) have been created. HDL files have been written for each operator found in the low level virtual machine (LLVM) compiler intermediate format and a table of area has been tabulated on Virtex-5 series supporting PR. The obtained partitions in LLVM-IR are read by a Perl script and converted into Verilog. Using PR design flow the partitions are bound to a PR region using PlanAhead SW. The results highlight the pros and cons of this technique by comparing the time required in SW and PR flow.

Keywords- Control flow graph, Low level virtual machine compiler, High level synthesis, Field programmable gate arrays, resource estimation, Partial reconfiguration.

I. INTRODUCTION

Accelerators are the applications which have been migrated from software to hardware implementation for increasing the performance. In contemporary chips examples of such accelerators are advance encryption algorithm (AES), Cyclic redundancy check (CRC), image processing etc. In such a migration, the HDL code is manually written to optimize it for application specific integrated circuits development (ASIC) [1]. The performance gain in this process is ten folds but the cost increases because of ASIC based design. The other way around is the automatic generation of HDL and implementation on FPGAs. Many academic and commercial compilers are available which can be used for automatic generation; examples include Spark, ROCCC, LegUp etc. [2]. FPGAs do not deliver as much performance as ASIC based design but chip fabrication can be by-passed. The automatic generated code for practical designs consumes large area of the device. Following the embedded design flow in Xilinx we concluded that an Intellectual property (IP) can be allocated 20% to 30% of the device area in Virtex-5 series. The application which consumes 60% cannot fit in such design. This data's can vary from chip to chip. Conventional method for putting such applications on FPGA is by partitioning the design into HW and SW parts [1]. In this research work we propose the design flow of HW partitions of the same applications. To run such application we can partition the design and produce partial HDL files. To partition the design we need to know the resources required for each operator. The partitions can then be mapped to partial reconfigurable regions in Planahead software. The scheduler to run the partitions can be written in the Software development kit (SDK) in Xilinx.

Research works published in this domain can be cited firstly in [3] and have shown the estimation of IO pins and time required for the benchmarks. The work is outdated as the density of logic elements has increased many folds.

The work done in the area estimation was first seen in [4]. The authors show the number of CLB consumed for a design based on number of operator, their bit-width and number of registers. The estimation is applicable to dataflow model where semantics is similar to netlist. The work in [5] shows the estimation model for Matlab based system generator designs. The model is well developed for Simulink based design and is not applicable to HLL. The work in [6] shows the compiler framework required for resource estimation of ANSI-C programs. Similar work has been done in later but synthesis has not been done for comparison.

The resource estimation require a mathematical model giving results taking inputs as operators, bit-width and types of operators and return the number of LUTs/registers used. This model has been thoroughly proposed in [7]. The results have only been shown for one benchmark which is not generic and estimation is not for ANSI-C programs. The current work shows the usage of resource estimation for ANSI-C program with a complete design flow.

There has been extensive research work in the field of high level synthesis which has produced many commercial tools. Since the objective of this paper is not HLS but showing the usage of HLS in a different way, literature survey has not been shown for it.

The remainder of this paper is organized as follows. Section II gives a design flow for implementing C applications on FPGA with resource estimation and partitioning process. Section III discusses the problem definition and its solution. Section IV shows the theoretical and algorithmic approach for resource estimation and library creation. Section V elaborates the high level synthesis. Section VI describes the comparative results. Section VII and VIII show future work and conclusion of our implementation respectively.

II. PROPOSED DESIGN FLOW FOR HW-HW PARTITIONING

Reconfigurable Computing Systems (RCS) offer a versatile platform for implementing applications in software or hardware using FPGAs. Additionally they offer what is known as partial reconfiguration which allows loading partial HW compiled bitstream that are stored in a memory and loaded with a help of a configuration controller [8]. This feature is also known as run-time reconfiguration (RTR), as the configuration is loaded when the systems are running. Fig.1 shows one such PRR region and two partially reconfigurable modules (PRM) named as *add* and *Mult* bound to this region.

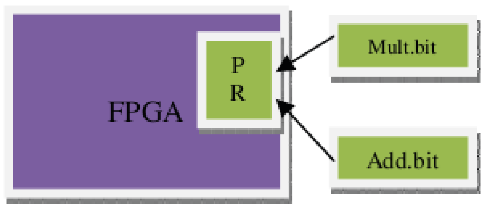


Fig. 1 FPGA based system with one PR region with two bit files mapped

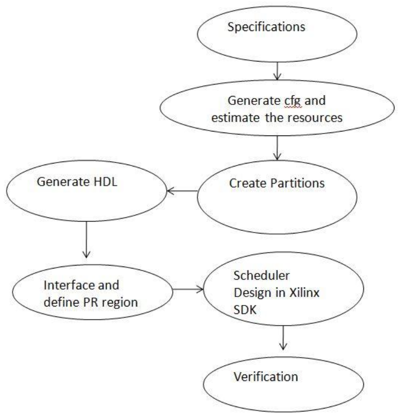

Specifications
Generate cfg and estimate the resources
Create Partitions
Generate HDL
Interface and define PR region
Scheduler Design in Xilinx SDK
Verification

Fig. 2 HW-HW partitioning design flow in Xilinx PR tool

The steps given below describe the proposed design flow for HW-HW partitioning as shown in Fig.2

The application written in C is converted to LLVM IR and is optimized for better HW generation. A library is created in HDL of instructions in LLVM IR and synthesized in Xilinx ISE. The output is tabulated in a table. The resource requirement of the program is then calculated and manual decision is taken from the results to do partitioning. The generated partitions are converted to Verilog by parsing the LLVM IR. A PR project is created in PlanAhead SW and generates the partial bit files are generated for each partition. Results are compared with respect to time results in SW and HW PR design flow.

III. PROBLEM DEFINITION AND GENERATING CONTROL FLOW GRAPH

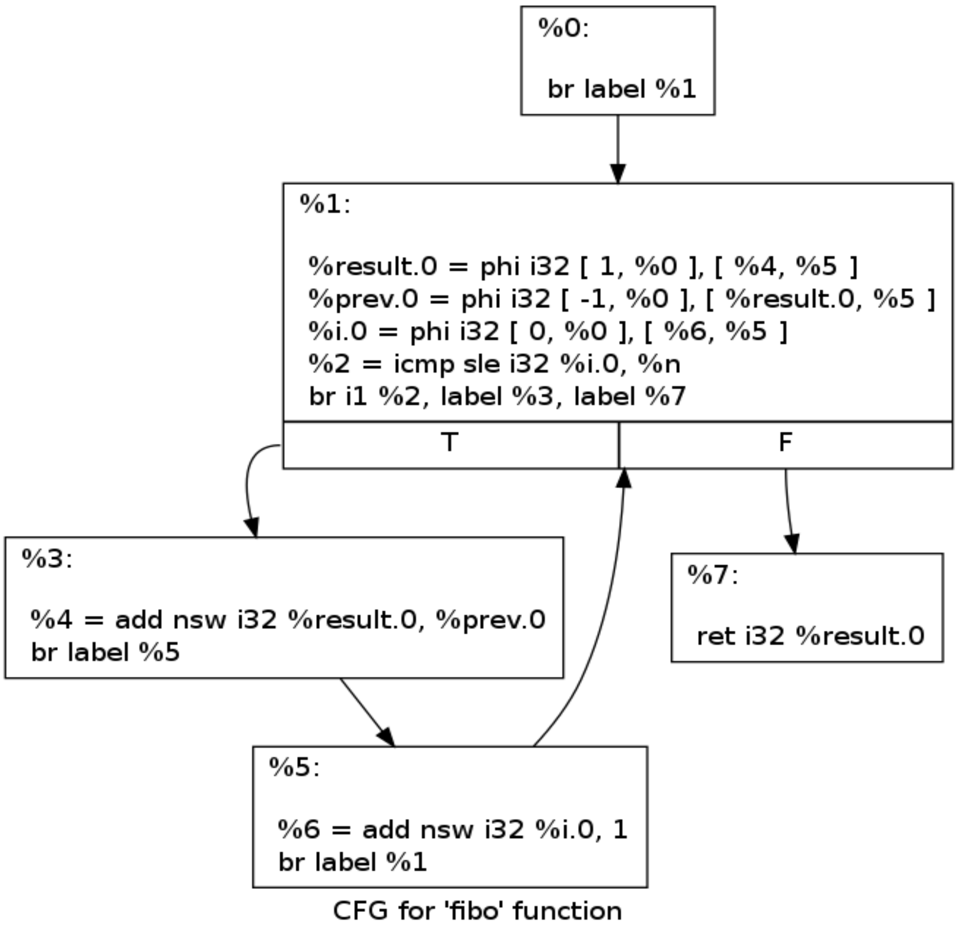
Suppose a C program that has three for loops, as they are ones that consume most of the time in the program. Assume when migrating this program to HW, the total resource estimation exceeds the amount that can be allocated. This requires that the program should be partitioned and executed in the correct order. Since there many options possible during the partitioning phase, it is necessary to systematize the process and make it robust. In this process it is necessary that we estimate the amount of resources that each line of C code will take when it converted to HW. Hence this places the requirement of parsing the C code using a compiler and generates operator level information. This is can be done using a LLVM compiler that has an easy instruction set and has many in-build optimization passes available.

A control flow graph (CFG) in literature is a representation, of all paths that might be traversed through a program during its execution. In compiler, a control flow graph node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets. The representation of a piece of code in control flow graph (CFG) is essential to many compiler optimizations and static analysis tools. CFG is an intermediate representation which carries the control and data flow. Control flow graphs have been frequently used for the automatic generation of HDL [9]. Compilers usually decompose programs into their basic blocks as a first step in the analysis process. Basic blocks from the vertices or nodes in a control flow graph.

Any compiler can be used to generate CFG [10] e.g. are SUIF, LLVM, GCC, TRIMARAN. The one having the in-built pass for CFG generation would be most optimum. This pass is available in LLVM [11], so it is a good candidate for generation of CFGs. Using LLVM we can generate the machine independent IR code and from that form the CFG of basic blocks. LLVM is a compiler infrastructure written in C++. It is designed for compile-time, link-time, run-time, and idle-time optimization of programs written in arbitrary programming languages. Many HLS compilers have used LLVM e.g. Ctoverilog, PandA, LegUp etc. *Clang* is the frontend of the LLVM complier that converts the C program into an Intermediate Representation (IR) that is similar to assembly language and useful for performing subsequent compiler stages. *Opt* utility allows various passes to run on the code for doing optimizations like dead code elimination etc. We have used mem2reg pass to minimize the memory access to minimum.

For Fibonacci, the c code and corresponding cfg is:

```
int fibo(int n){
int prev = -1;
int result = 1;
int sum;int i;
for(i = 0;i <= n;++ i)
{
sum = result + prev;
prev = result;
result = sum;
}
return result;}
```



```
%0:

 br label %1
```

```
%1:

 %result.0 = phi i32 [ 1, %0 ], [ %4, %5 ]
 %prev.0 = phi i32 [ -1, %0 ], [ %result.0, %5 ]
 %i.0 = phi i32 [ 0, %0 ], [ %6, %5 ]
 %2 = icmp sle i32 %i.0, %n
 br i1 %2, label %3, label %7
```

```
%3:

 %4 = add nsw i32 %result.0, %prev.0
 br label %5
```

```
%7:

 ret i32 %result.0
```

```
%5:

 %6 = add nsw i32 %i.0, 1
 br label %1
```

Fig. 3 CFG of Fibonacci series

IV. RESOURCE ESTIMATION AND LIBRARY CREATION

For estimating the amount of chip space required for each operator a library is required. The table-1 shows the LLVM instructions and their corresponding area. The Xilinx FPGA Virter-5 series xc5vfx70t-1ff1136 [12] on ML507 was used to compile the operators. This kit was chosen as it offers Partial Reconfiguration (PR) design.

Table-1

llvm instruction set: i32	Equivalent hardware unit	LUT	FF
Phinode	Multiplexer +register	32	32
Phinode	Up/down counter Or acculumator	33	32
add	Adder	32	-
fadd	FP Adder	**721**	-
sub	Substractor	32	-
fsub	FP Sub	874	
mul	Multiplier	3/128 DSP	-
fmul	FP Multiplier	49	-
icmp_eq	equal	11	-
icmp_ne	Not equal	11	

Since the datapath contains arithmetic operators and consume most of the area, there estimation is of prime concern. In this work we are focusing on area hence we have not shown the delay values. In the table, column 1 is the LLVM instruction, column 2 is the equivalent hardware unit and column 3 is the area in terms of Look-up tables (LUTs). In order to estimate resources correctly we need so see the types and quantity of resources available in Xilinx Virtex-5. We need to inspect the interrelation in the result summary of the Xilinx ISE to create a formula. The various kinds of resources available in Virtex-5 are slice registers with LUTs and flip-plops, DSP slices and BRAMs. We need to create formula for slices registers and LUT which are the primary resources under the assumption, that no operator sharing i.e. one to one mapping for each instruction in CFG to its corresponding hardware block.

The total estimated area depends on the area consumed by three files FSM, datapath and top level. The area consumed by FSM and top level is insignificant. The analysis is given below:

$\text{Estimated}_{\text{Total}} = \text{FSM}_{\text{area}} + \text{Datapath}_{\text{area}} + \text{Toplevel}_{\text{area}}$

Proposed theatrical formula for resource estimation:

Registers = (number of phinodes) * (width of variable) + [log(number of nodes)/log2]

LUT slices = (LUT slices of hardware) * (occurrences) + LUT slices used in FSM

Since datapath is 32 bit and the FSM signals are 1 bit wide, LUT slices used by FSM can be ignored without losing much in accuracy. Based on the resource table,

Case a : Fibonacci series program,
Expected slice register = 3*32+3 = 99
Expected LUTs = 2*(phi node) + 1*(constant increment phinode) + 1*(32-bit adder) + 1*(icmp_sle) = 161

Case b: GCD program,
Expected slice register = 4*32+3 = 131
Expected LUT = 4*(phi node) + 2*(32-bit subtractor) + 1*(icmp_eq) + (icmp_sgt) = 236

Case c: factorial program,
Expected slice register=4*(phinode)+4=132
Expected slice LUT=1*(phi node) + 2*(constant increment phinode) + 2*(icmp_sle) = 194

The optimized intermediate code from LLVM is given as input to a Perl file which processes the IR code and outputs a text file containing the resource required for each basic block and the total resource requirement. The algorithm used for the resource estimation is given below

ALGORITHM 1. RESOURCE ESTIMATION ALGORITHM

```
Create library of resources
No.of LUTs = 0
No.of DSP slices = 0
For each line in LLVM IR
{
If (line==BasicBlock name)
{Make a new entry in the Resource requirement table
LUTs required for the previous block = no. of LUTs
DSP slices required for the previous block = no. of DSP slices
No.of LUTs=0
No.of DSP slices=0}
Else {
Search for matches with library
If a match is found {
No.of LUTs += Matched entry's resource requirement in the library
No.of DSP slices += Matched entry's resource requirement in the library}
}
}
Total LUTs required = Sum of LUTs required for each block
Total DSP slices required = Sum of DSP slices required for each block
END
```

V. HIGH LEVEL SYNTHESIS

High-level synthesis [13], is the design flow to obtain hardware automatically from high level specification. The high level specification can be in various forms e.g. flowchart, C, C++, pseudo code etc. These specifications are sequential in behavior hence the flow is converted to concurrent execution in hardware. Parallelism is extracted and clock is added to the generated hardware. The generated hardware is in RTL form described in Verilog or VHDL. The goal of HLS is to let hardware designers efficiently build and verify hardware, by giving them better control over optimization of their design architecture, and through the nature of allowing the designer to describe the design at a higher level of tools while the tool does the RTL implementation. Verification of the RTL is an important part of the process. With embedded system design flow the IP is interfaced to the bus and bus controller signals are wrapped around it. The master i.e. processor is responsible for initiating the transactions and computation is handled by IP. A Perl script was written to parse the LLVM IR and convert to Verilog.

The Perl file gives three files as outputs FSM, data-path and the top module. The FSM module preserves the dataflow dependency and data-path preserves the dataflow. The top module instantiates the two modules and bind them together.

The LLVM IR instructions can be divided into four categories [14].

1. Datapath instructions e.g. add, sub, mul,
2. Conditional jumps e.g. whilecond, ifcond.
3. Control flow e.g. br, ret
4. PHI nodes define the incoming branch.

Fig. 4 depicts the generated data-path, FSM and top module for the CFG in fig. 2. Each basic block is modeled as one FSM state. Each phi node defines the incoming branch which infers a MUX operation. The first line in %1 state means assign the variable *%result.0* a value of 1 from *%0* state or a value of *%4* from *%5* state. The select line of the MUX comes from FSM when in state *%1*.The output of the MUX goes to a register which has an enable line from FSM. The phi node with loop variable infers a counter in synthesis, so in diagram for control variable with constant adder is shown. The operators like *icmp* and *add* are synthesized as separate blocks. The output of the comparator is send to FSM to update the state. In state *%7* the program ends and result is the output. Fig. 3 shows the complete block diagram of the Fibo. module. Signals generated in FSM are mapped to the data-path in the top level module. As shown in fig. 2 the CFG has five states, so are in FSM diagram. The various enable signals are generated when the state changes. Since this is first step for resource estimation, simple programs with no high level constructs have been used. There are many limitations that this HLS has like restricted subset of C (no function calls, memory access). In future we tend to remove these restrictions.

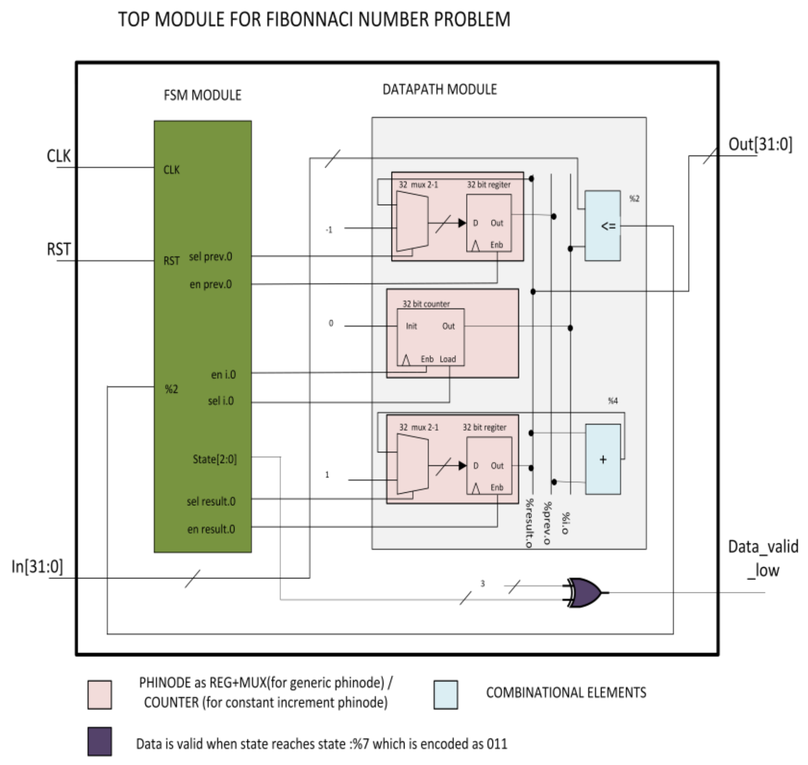


Fig. 4 Top Module for Fibo program

The actual resource consumption is measured by no. of LUT flip-flop pair used. Generally, as seen from table, number of LUT slices used is larger in number than FF used. So to save resource usage on FPGA slice LUT utilization must be minimized. Table-2 shows the results of the resources used by the program after synthesis and theoretical calculation. The results are also compared with a commercial tool Vivado.

Table-2 Comparison of Resources

Module(1)	**Estimate(2)**		**Actual after synthesis(3)**		**Vivado Results**
	Slice Lut	Slice register	Slice Lut	Slice register	Slice Lut, registers
Nth Fibonacci number	161	99	164	99	168,96
Gcd function	236	131	240	134	212, 65
Sum of all factorials upto ns	194 + 3/128 DSP	132	199 + 3/128 DSP	136	265 + 4/128 DSP,163

VI. RESULTS

The LLVM IR can be extracted with different optimizations for the convenience of the programmer. It is found that different optimizations give different area. For this purpose, three C programs are taken as example.

1. GCD of two numbers,
2. Sum of Fibonacci series upto n numbers,
3. Factiorial of a given number.

Different optimizations used for analysis are:

Optim1: -mem2reg, -instsimplify
- mem2reg pass considers memory as register
- instsimplify simplifies instruction and inserts phi nodes

Optim2: - mem2reg –lcssa -licm
- lcssa is loop closed SSA form pass. It places phi nodes at the end of loops
- licm is loop invariant code motion. This pass identifies the statements which are inside the loop whose values are not changing and keeps them outside the loop

Optim3: -mem2reg -loop-rotate -loop-reduce
- loop-rotate rotates loop and –loop-reduce reduces the strength of array references inside loops

Optim4: -mem2reg -loop-unswitch
- loop-unswitch creates multiple loops wherever it is necessary

Optim5: -mem2reg -loop-rotate -loop-unroll
- loop-unroll unrolls the loop. Here the unroll count used is 10.

Table-3 Shows the LUT/DSP resource estimation for each of the optimizations

Function	Optim1	Optim2	Optim3	Optim4	Optim5
Gcd	236/0	268/0	311/0	268/0	1859/0
Factorial	320/6	384/6	512/6	384/6	1376/60
Sum of Fibonacci series	192/0	224/0	288/0	224/0	1152/0

Fig. 5 below is the graphical representation of LUTs used for different optimizations. It is clear that loop unrolling increases the resource requirement. But with loop unrolling, concurrency can be achieved.

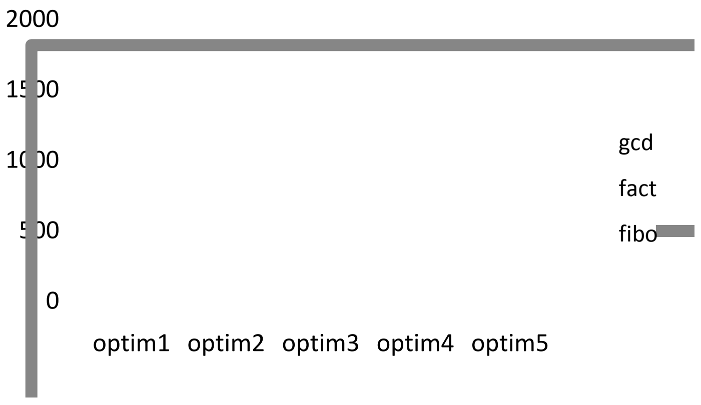


Fig.5 Comparison of optimizations on HLS

The Verilog codes which are generated after is synthesized using Xilinx ISE. The output waveforms generated for the GCD program is in Fig. 6.

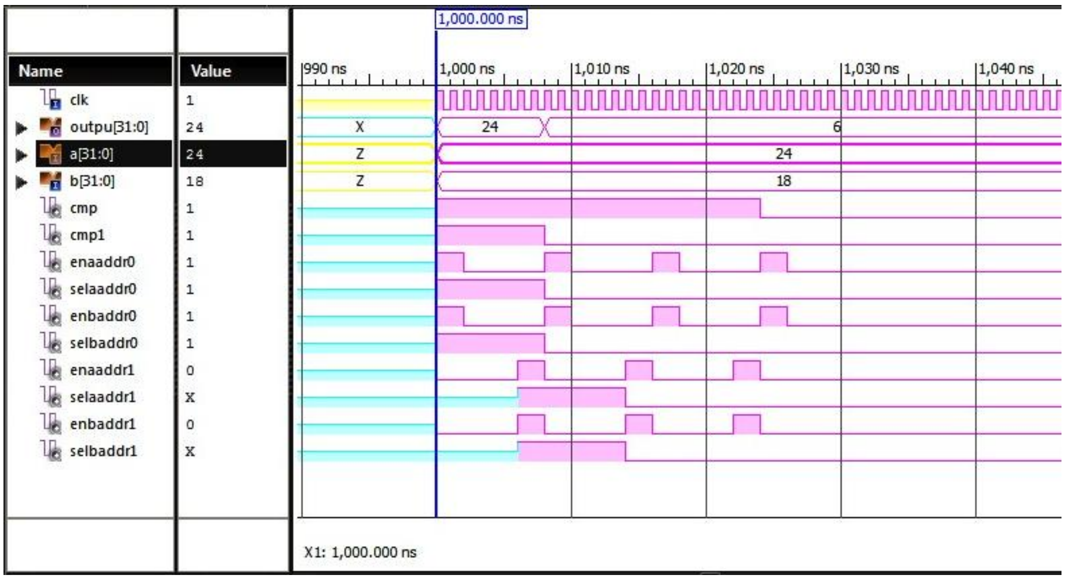


Fig. 6 Verification of the GCD program

An experimental setup consists of a PowerPC running at 100 MHz, ICAP controller at 100 MHz and xps timer at 100 MHz. The three partitions were interfaced to the PLB bus and partial bit files were generated in PlanAhead tool. SDK program was written to load the files dynamically. The partial bitstream files were placed in flash memory. As an example, the area of DCT program is calculated. The callgraph of DCT program is given in figure 7. The table 4 shows the amount of LUTs/DSP slices required for a DCT program. The taken to run the DCT using partial reconfiguration was calculated using xps_timer running at 100 MHz. A data set of 64x 64 values was given as input and configuration of F and Y-C was done as two configurations. Time was found to be to 0.163 sec for each configuration and total time was 0.429 when the entire design was running.

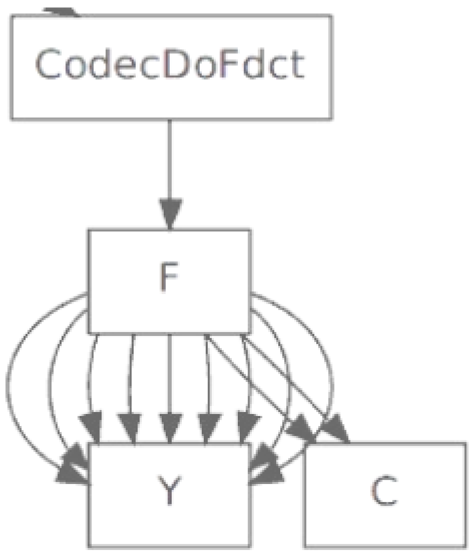


Fig. 7 C program for Discrete Cosine transform with three functions.

Table-6 Resource table and Reconfiguration time taken

Function	LUTs required	DSP Slices required
CodecDoFdct	582	0
Y	1941	4
F	6582	60
C	11	0

VII. FUTURE WORK

In this work we were able to estimate the resource requirement of the program on the reconfigurable hardware. Depending on the estimated values the program can be now partitioned into clusters and executed on partial reconfigurable HW. By estimation technique we will be able to create cluster of the required size For mapping the partitioned design to one PR region a wrapper generation is requires which will interface to the bus. This wrapper should be automatically generated for each partition created. The scheduler was designed statically for the program; this is a major bottle neck in design flow. A significant research work is required to generate the scheduler aromatically depending on the control flow of the program. No such scheduler has ever been discussed in literature although many works in the direction still remain. The current work will continue in this direction and bring up a robust scheduler

VIII. CONCLUSION

In this work we successfully showed the process of resource estimation by creating library. We have verified the formula by generating HDL code and synthesizing on Xilinx. The results show that the proposed design flow is very useful extension to currently existing tools on partial reconfiguration and HLS synthesis that will allow any program to migrate on FPGA irrespective of amount of resources it can use. The reconfiguration process is still not very robust as it gives error at many places where the placement is not routable. The reconfiguration time was high as the loading of bitstream from the ICAP controller is limited by its bit with and clocking. This time can be further improved by placing small bitstreams in BRAM memory which will be local to the chip.

REFERENCES

[1] Klein R.A., Moona R., "Migrating software to hardware on fpgas", International conference on field programmable technology, pp. 217-224, December, 2004.

[2] Kuon I., Rose J., "Measuring the gap between FPGAs and ASICs", IEEE transactions on computer-aided design of integrated circuits and systems, vol. 26, no. 2, February, 2007.

[3] Min Xu,. Kurdahi F.J," A Tool for Chip Level Area and Timing Estimation of Lookup Table Based FPGAs for High Level Applications ", Design Automation Conference, pp.435-440, January, 1997.

[4] Nayak A., Haldar M., Choudhary A., Banerjee P., "Accurate Area and Delay Estimators for FPGA", Design, Automation and Test in Europe Conference and Exhibition, pp.862-869, March, 2002.

[5] Changchun Shi S., Hwang J., McMillan S., Root A., Singh V.,"A System Level Resource Estimation Tool for FPGA", Field Programmable Logic and Application,Lecture Notes in Computer Science Volume 3203,, pp 424-433, September, 2004

[6] Kulkarni Z., Najjar W., Rinker R., Kurdahi F.J., "Compile-Time Area Estimation for LUT-Based FPGAs", ACM Transactions on Design Automation of Electronic Systems, Volume 11 Issue 1, January, 2006.

[7] Arce-Nazario R. A., Juan S., Jimenez M., Rodreguez D., "Architectural Model and Resource Estimation", Reconfigurable Computing and FPGAs International Conference, pp. 103-108, December, 2008.

[8] Bobda C., Introduction to Reconfigurable Computing: Architectures, Algorithms and Applications, Springer, 2007.

[9] Patrick R. Schaumont, A practical introduction to hardware/software codesign, Springer international edition, 2011

[10] Giovanni De Micheli, Synthesis and optimization of Digital Circuits, Tata McGraw-Hill Edition, 2003.

[11] Low level vitrual machine,http://llvm.org/, last accessed on June, 2015.

[12] http://www.xilinx.com/products/boards/ml507/reference_designs.htm

[13] Cong J, Liu B., Neuendorffer S., Juanjo N., Kees V., Zhang Z., "High-level synthesis for fpgas: from prototyping to deployment", IEEE transactions on computer-aided design of integrated circuits and systems, vol. 30, no. 4, April, 2011.

[14] Jun.-Prof. Dr. Christian Plessl, Architecture Synthesis, Lecture notes, 2012.