A Method for Power Aware Compilation: RISC-V Instruciton Set Architecture

June 2, 2015

1 Abstract:

This thesis deals with the method and the extent to which Dynamic Power consumption can be reduced by optimization done to the compiled code of RISC-V ISA. The optimization in this method is done at post copilation level. This method uses the RTL of the Target machine to decrease the dynamic power and so it is machine dependent as opposed to machine independent compiler optimizations. Though this thesis deals with the implementation of the method on RISC-V ISA, it could very well be implemented on any other ISA.

2 Introduction:

Semiconductor technology is advancing every year, evolving and giving rise to more compact, faster solutions. This is still best described by Moore's Law:

> "*the number of transistors per square inch, in a dense integrated circuit will approximately double every two years*"- *Gordon Moore*

With this rapid growth in technology, chip capacity has grown very fast and SOC(System-On-Chip) has become very crucial to integrate hardware, applications and software. Thus began the popularity of Embedded Systems in multimedia and wireless applications. Most of these embedded solutions are portable.Hence reducing the energy dissipation without performance overhead has become a challenge.

Lesser Area and higher performance leads to high power dissipation. Power management through techniques like Dynamic Voltage Scaling by Under-volting,

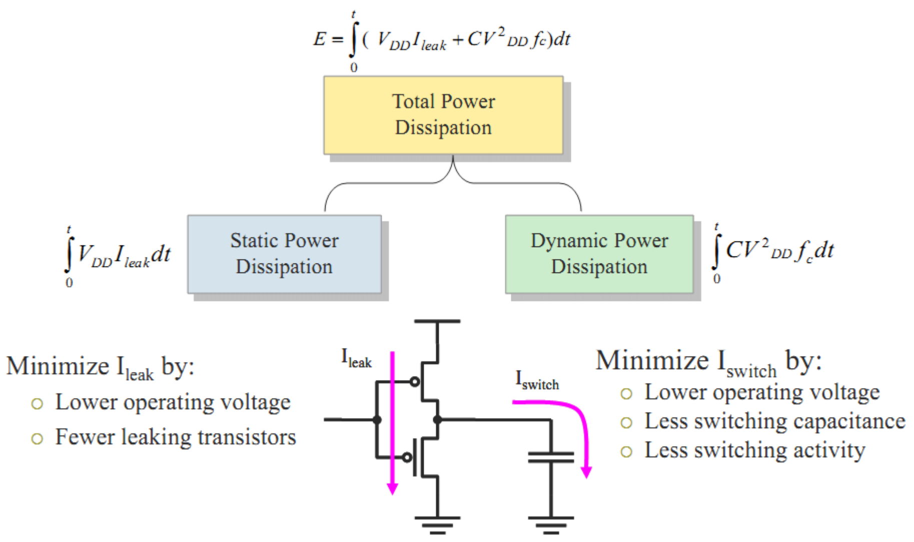

$E = \int_0^t (V_{DD} I_{leak} + CV^2_{DD} f_c) dt$
Total Power Dissipation
$\int_0^t V_{DD} I_{leak} dt$
Static Power Dissipation
Dynamic Power Dissipation
$\int_0^t CV^2_{DD} f_c dt$
Minimize I_{leak} by:
Lower operating voltage
Fewer leaking transistors
I_{leak}
I_{switch}
Minimize I_{switch} by:
Lower operating voltage
Less switching capacitance
Less switching activity

Figure 1: CMOS Power Dissipation

or, Dynamic Frequency Scaling are ways to tackle the power issue but they come short in terms of maintaining the performance. Thus, simply scaling voltage or frequency to reduce power is insufficient as the desired performance level cannot be achieved. Hardware and software solutions that maintain performance while reducing power consumption are required. This paper discusses a methodology in this regard.

Power consumption in semiconductors can be divided into two categories in Low Power design.Static dissipation, on one hand, is contributed by sub-threshold conduction through OFF transistors and leakage through reverse-biased diodes. Dynamic dissipation is due to charging and discharging of load capacitances and short circuit current while both PMOS and NMOS networks are partially ON.

CMOS circuits are designed theoretically to not consume any power if their inputs are not switching (or in the case of a latch, if there is no clock input. In reality, all FET transistors leak current between the source and drain (if there is a voltage potential across the source-drain) even if the gate voltage is in the off position. This current is called sub threshold current and used to be insignificant. However, as devices have become smaller and smaller and operating voltages have not scaled down with technology scaling (due to problems scaling the threshold voltage), this sub threshold voltage is becoming an increasingly important component of the total power of a chip.This becomes especially true in devices that have millions of these simultaneously leaky circuits, even if the device is not doing anything. Early power managed devices (like microprocessors) were able to control power consumption in sleep modes by simply slowing or stopping the clock (known as clock gating) but as static power consumption has increased, it has become more important to consider techniques such as voltage islands on chips where the voltage to large portions of a chip can be shut off (or reduced) to reduce the static leakage currents.

The dynamic power is the power associated with switching. Wires and inputs to circuits (which are usually gate electrodes) can be modeled as capacitive loads to the circuit. When the circuit has to switch from a high voltage to a low voltage (or visa versa) then this capacitance has to be charged or discharged. This takes a certain amount of energy and if you repeat this billions of times every second, it becomes a continuous or AC power. There can also exist in circuits an intermediate position where in the process of switching, there can appear a direct path in the circuit between voltage and ground which can cause a switching current that is resistive and therefore different than the capacitive power just described. This resistive current can be managed with non-overlapping clocks in latch design, but this power may also be considered an dynamic power as it only occurs when circuits are switching.In logic design, dynamic power increases whenever logic toggles from 1 to 0 or 0 to 1. And we will be concentrating on reducing this Dynamic Power as this is the major contributor for power dissipation in CMOS systems.

3 Power Aware Compilation

Energy Aware systems, that is systems that try to control it's power consumptions are common in today's system design. Every code designed for a specific set of operations consume energy for its computation. Hence leads to the necessity to optimize the code from an energy consumption perspective.There are two design methodologies to reduce energy consumption in embedded computing systems

- *Hardware Approach:* This method involves checking leakage current in the integrated circuits, designing alternatives that reduce heat dissipation namely Register assisted techniques, Low Power cache, Partitioned Data Cache.

Other techniques include Register spilling, Memory access due to cache misses.

There are already fair amount of traditional optimizations built into compilers like Instruction scheduling, Register Allocation, Loop optimizations etc. All these optimizations are independent of the target machine and hence would miss out on better machine specific software optimizations. The methodology dealt with in this thesis is a software optimization technique that avails the hardware specifications of the target.

- *Software Approach*: While hardware optimizations has been the primary focus of multitude of studies and are fairly advanced, software approaches to optimizing power are relatively young. Progress in understanding the impact of traditional compiler optimizations and developing new power-aware compiler optimizations are important to overall system energy optimization. Software has a significant impact on the overall energy consumption being the main determinant for activity on the processor core, interconnect and memory system, which are, collectively, responsible for significant percentage of total power dissipation. Despite this observation, to date, most of the compiler techniques consider only delay and area as their main performance metrics. With the growing demand for power-aware software, there is an acute need for investigating energy-oriented compilation techniques and their interaction and integration with performance-oriented compiler optimizations.

3.1 Methodology:

In logic design, dynamic power increases whenever logic toggles from 1 to 0 or 0 to 1. Hence, we would be targeting on reduction of this Dynamic Power by decreasing the total number of toggles in the processor during computation of the code. The optimization would not be done at the compiler level, or on High Level Language. Instead it is done at post-compilation level on the raw machine code that is generated.

The object code generated after compilation is a set of instructions that are specific to the architecture of the hardware. These instructions are part of the ISA upon which the hardware is designed. Given a code to be optimized, the order of these set of instructions is not completely bound. The order of two instructions can be changed as long as there is no dependency between them.

There are two types of dependencies between instructions that govern their order of execution.They are Data dependency and Control dependency.

A data dependency is a situation in which a program statement or instruction refers to the data of a preceding statement.Recklessly executing multiple instructions without considering related dependences may cause danger of getting wrong results, namely hazards.Three cases exist:

1. Flow (data) dependence: This dependency occurs where an instruction refers to a result that has not yet been calculated or retrieved. This can occur because even though an instruction is executed after a previous instruction, the previous instruction has not been completely processed through the pipeline.

2. Anti-dependence: An anti-dependency, also known as write-after-read (WAR), occurs when an instruction requires a value that is later updated.

3. Output dependence: An output dependency, also known as write-after-write (WAW), occurs when the ordering of instructions will affect the final output value of a variable.

Consider two instructions i and j, with i occurring before j. The possible data dependencies are:

- RAW (read after write) - j tries to read a source before i writes it, so j incorrectly gets the old value.

- WAW (write after write) - j tries to write an operand before it is written by i. The writes end up being performed in the wrong order, leaving the value written by i rather than the value written by j in the destination.

- WAR (write after read) - j tries to write a destination before it is read by i , so i incorrectly gets the new value.

Intuitively, there is control dependence between two statements S1 and S2 if

- S1 could be possibly executed before S2

- The outcome of S1 execution will determine whether S2 will be executed.

After performing dependency analysis on this code, we will generate all the possible permutations of the program that are valid , that is they do not compromise the integrity of the code. Each such permutation will be run on the RTL of the hardware and the power consumption will be estimated based on the number of toggles. The best among all these permutations would be chosen to be loaded into the embedded processor which will execute the code many multiples of times, saving power enormously. Though this idea seems simple there are many intricacies involved which will be further discussed as we go along. As it can be observed, this methodology is neither bound by any ISA nor by any RTL. Given a way to implement it on a specific ISA, it can be applied to any RTL of the ISA. But for the sake of convenience this paper is limited to such analysis on the RISC-V Architecture.

4 RISC-V ISA:

RISC-V is an open source implementation of a reduced instruction set computing (RISC) based instruction set architecture (ISA). An instruction set is not a computer design, but a description of the bit-patterns of the instructions for a computer. The ISA defines the boundary between the electronics and the software. The ISA is one of the most important interfaces in a computer system. Most ISAs are commercially protected by patents, preventing practical efforts to reproduce the computer systems. In contrast, RISC-V is open, permitting any person or group to construct compatible computers, and use associated software.

RISC-V has 32 integer registers and can have 32 floating-point registers. The memory is addressed by 8-bit bytes. The instructions must be aligned to 32-bit addresses. Register number 0 is a constant 0. The assembler uses register 0 as a place-holder to make any of several human-readable instructions into one machine instruction. The only instructions that access main memory are loads and stores. All arithmetic and logic operations occur between registers. The instruction set includes other features to increase a computer's speed, while reducing its cost and power usage. These include placing most-significant bits at a fixed location to speed sign-extension, and a bit-arrangement designed to reduce the number of

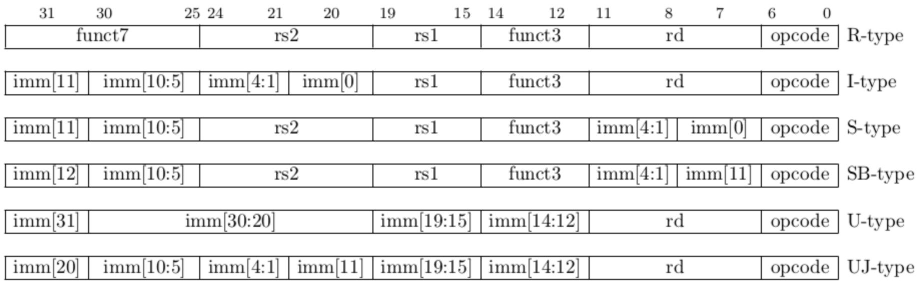

31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11] | imm[10:5] | imm[4:1] | imm[0] | rs1 | funct3 | rd | opcode | I-type
imm[11] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[0] | opcode | S-type
imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | SB-type
imm[31] | imm[30:20] | imm[19:15] | imm[14:12] | rd | opcode | U-type
imm[20] | imm[10:5] | imm[4:1] | imm[11] | imm[19:15] | imm[14:12] | rd | opcode | UJ-type

Figure 2: RISC-V base instruction formats showing immediate variants

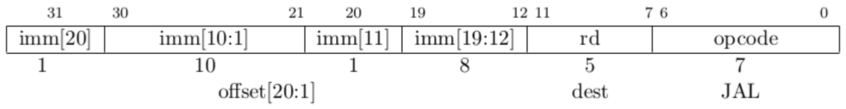


Figure 3: JAL Instruction for RISC-V ISA

multiplexers in a CPU.

RISC-V intentionally lacks condition codes, and even lacks a carry bit.Instead RISC-V builds comparison operations into its conditional-jumps.Use of comparisons may slightly increase its power usage in some applications. The lack of a carry bit complicates multiple-precision arithmetic. RISC-V does not detect or flag most arithmetic errors, including overflow, underflow and divide by zero.RISC-V also lacks the "count leading zero" and bit-field operations normally used to speed software floating-point in a pure-integer processor.

In the base ISA, there are four core instruction formats (R/I/S/U), as shown in Figure. All are a fixed 32 bits in length and must be aligned on a four-byte boundary in memory. An instruction address misaligned exception is generated if the pc is not four-byte aligned on an instruction fetch.

Unconditional Jumps The jump and link (JAL) instruction uses the UJ-type format, where the J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the pc to form the jump target address. Jumps can therefore target a ±1 MiB range. JAL stores the address of the instruction following the jump (pc+4) into register rd. The standard software calling convention uses x1 as the return address register. Plain unconditional jumps (assembler pseudo-op J) are encoded as a JAL with rd=x0.

31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	0	dest	JALR

Figure 4: JALR Instruction for RISC-V ISA

31	30 — 25	24 — 20	19 — 15	14 — 12	11 — 8	7	6 — 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12,10:5]		src2	src1	BEQ/BNE	offset[11,4:1]		BRANCH
offset[12,10:5]		src2	src1	BLT[U]	offset[11,4:1]		BRANCH
offset[12,10:5]		src2	src1	BGE[U]	offset[11,4:1]		BRANCH

Figure 5: Conditional Branch Instructions

The indirect jump instruction JALR (jump and link register) uses the I-type encoding. The target address is obtained by adding the 12-bit signed I-immediate to the register rs1, then setting the least-significant bit of the result to zero. The address of the instruction following the jump (pc+4) is written to register rd. Register x0 can be used as the destination if the result is not required.

All branch instructions use the SB-type instruction format. The 12-bit B-immediate encodes signed offsets in multiples of 2, and is added to the current pc to give the target address. The conditional branch range is ±4 KiB.

Branch instructions compare two registers. BEQ and BNE take the branch if registers rs1 and rs2 are equal or unequal respectively. BLT and BLTU take the branch if rs1 is less than rs2, using signed and unsigned comparison respectively. BGE and BGEU take the branch if rs1 is greater than or equal to rs2, using signed and unsigned comparison respectively. Note, BGT, BGTU, BLE, and BLEU can be synthesized by reversing the operands to BLT, BLTU, BGE, and BGEU, respectively.

Load and store instructions transfer a value between the registers and memory. Loads are encoded in the I-type format and stores are S-type. The effective byte address is obtained by adding register rs1 to the sign-extended 12-bit offset. Loads

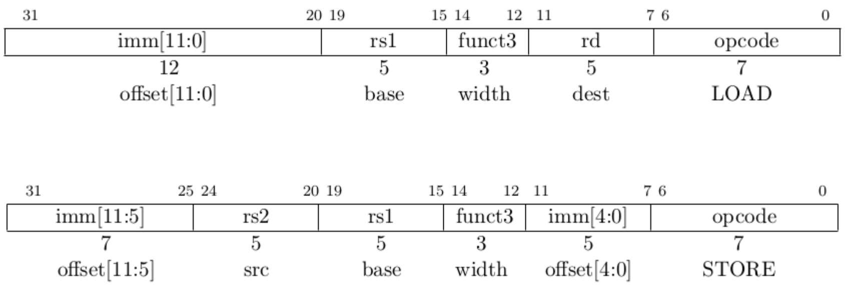


Figure 6: Memory Instructions in RISC-V ISA

copy a value from memory to register rd. Stores copy the value in register rs2 to memory.

The F (Floating Point) extension adds 32 floating-point registers, f0–f31, each 32 bits wide, and a floating-point control and status register fcsr, which contains the operating mode and exception status of the floating-point unit.The rest is similar to that of the Integer base instructions.

4.1 RISC-Tool Chain :

The RISC-V toolchain is a standard GNU cross compiler tool-chain ported for RISC-V. You will use riscv-gcc to compile, assemble, and link your source files. riscv-gcc behaves similarly to the standard gcc, except that it produces binaries encoded in the RISC-V instruction set. These compiled binaries can be run on spike, the RISC-V ISA simulator. They can also be used to generate a hexadecimal list of machine code instructions that can be loaded into the instruction memory of a simulated (or real) processor.

A cross compiler is used to generate the compiled code specific to the particular ISA, as mentioned in the earlier sections. A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the computer is running. *"risc-v gcc"* is the cross compiler tool used in this project to produce the compiled code specific to RISC-V ISA.

The RISC-V ISA specification defines the desired behavior of a RISC-V processor. The RISC-V Rocket core, a micro-architecture developed by the Berkeley Architecture group that implements the 32-bit instruction format of the RISCV64 ISA .

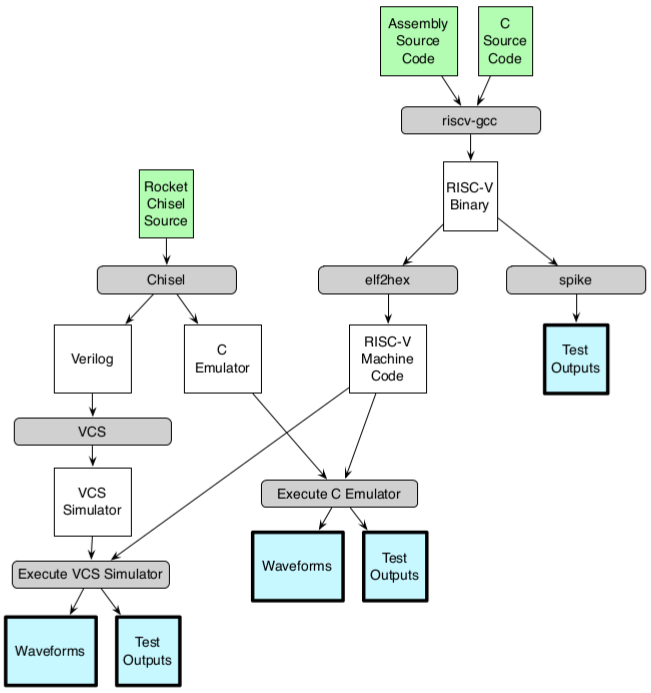


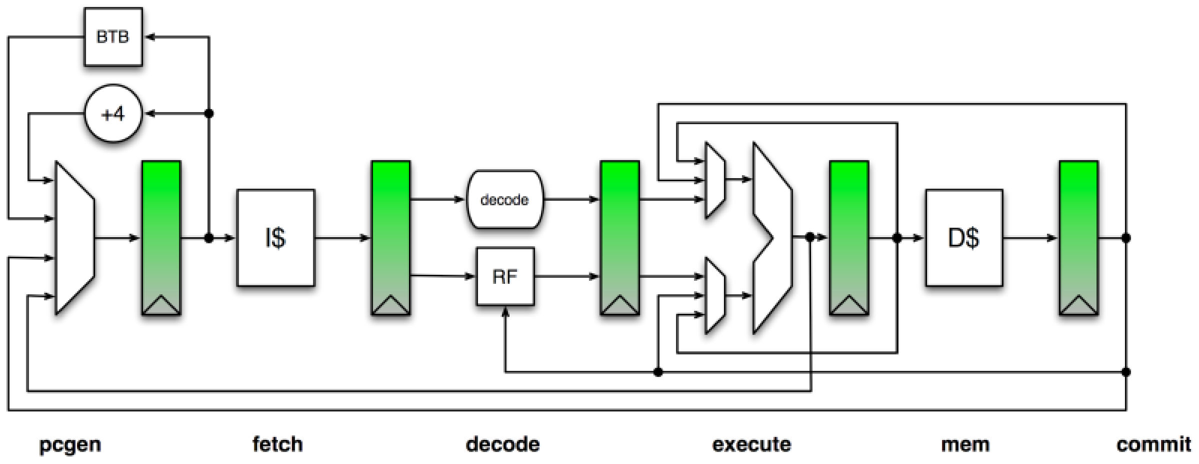
Figure 7: RISC-V Tool Flow

VCS compiles Verilog source files into a native binary that implements a simulation of the Verilog design. VCS can simulate both behavioral and RTL level Verilog modules. In behavioral models, a module's functionality can be described more easily by using higher levels of abstraction. In RTL descriptions, a module's functionality is described at a level that can be mapped to a collection of registers and gates.

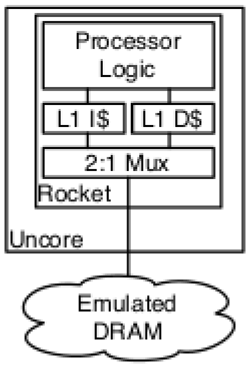
4.2 Rocket Core :

Rocket core is used as the Target Machine for all the simulations. Rocket is a 6-stage single-issue in-order pipeline that executes the 64-bit scalar RISC-V ISA. Rocket implements an MMU that supports page-based virtual memory and is able to boot modern operating systems such as Linux. Rocket also has an optional IEEE 754-2008-compliant FPU, which implements both single- and double-precision floating-point operations, including fused multiply-add. To instantiate a Rocket core, Rocket chip generator found in the rocket-chip git repository is used.

Rocket is an in-order, single-issue scalar processor that includes a six-stage integer pipeline. It has a 31-entry, 64-bit register file and uses a scoreboard to detect data hazards involving instructions with multi-cycle latencies. The processor has both a user and a supervisor mode; an synchronous trap or external (asynchronous) interrupt can trigger a transition from user to supervisor mode.



The Rocket core contains a fast L1 instruction cache and L1 data cache. In Rocket, these caches communicate through a simple bus to a simulated DRAM that acts as main memory for the system.



5 Algorithm :

The Algorithm optimizes the RISC-V Machine code that is generated after the compilation of the source code from Fig7. After the optimization, the new code retains it's functionality. This Algorithm is a step by step optimization of independent blocks of the machine code, where there is a linear transfer of program control flow between each block, called Basic Blocks. Each basic block goes through a dependency analysis to generate a Directed Acyclic graph. This DAG is then topologically sorted to get a level ordering. Maintaining this level order, a rearrangement of this machine code (one basic block at a time) which generates a lower number of toggles, is then chosen with the help of simulations done on the target processor. Hence the best way to describe this Algorithm is that it is a Machine dependent Optimization.

5.1 Construction of Basic Blocks :

In the first stage of the Algorithm, the input machine code is converted into a set of basic blocks. A basic block is a sequence of consecutive statements in which control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. So, The code in a basic block has:

- One entry point, meaning no code within it is the destination of a jump instruction anywhere in the program.
- One exit point, meaning only the last instruction can cause the program to begin executing code in a different basic block.

The input is in hex where each line has four instructions. It is the standard format in which the rocket-chip simulation accepts the inputs. The basic blocks are generated by first, identifying the leaders in the code. Leaders are instructions which come under any of the following 3 categories :

- The first instruction is a leader.
- The target of a conditional or an unconditional branch/jump instruction is a leader.
- The instruction that immediately follows a conditional or an unconditional branch/jump instruction is a leader.

Then starting from a leader, the set of all following instructions until and not including the next leader is the basic block corresponding to the starting leader. For this each and every instruction is decoded and leaders are identified.

Once all the basic blocks are generated, each line in the basic block corresponds to one instruction in binary.

from the basic blocks generated a Flow Graph is constructed. A flow graph is a directed graph. The nodes in the flow graph are Basic Blocks. There is an edge from B1 to B2 if and only if B2 immediately follows B1 in some execution sequence. If B2 immediately follows B1 in program text there is a jump from B1 to B2.B1 is a predecessor of B2, B2 is a successor of B1.

The code to implement this part, for convenience in text formatting, is written in Perl.

5.1.1 Implementation:

```
#!/usr/bin/perl
print "$ARGV[0]\n" ;
open(FH, "<$ARGV[0]") or die "Couldn't open file file.txt, $!";
open(FH1, ">block.hex") or die "Couldn't open file file.txt, $!";
while($line = <FH>){
   chomp($line);
   $temp1 = $line;
   $temp1 =~ s/^[0-9,a-f]{16}//;
   $temp_1=$temp1;
   $temp_1 =~ s/^[0-9,a-f]{8}//;
   $temp_2=$temp1;
   $temp_2 =~ s/[0-9,a-f]{8}$//;
   $temp2 = $line;
   $temp2 =~ s/[0-9,a-f]{16}$//;
   $temp_3=$temp2;
   $temp_3 =~ s/^[0-9,a-f]{8}//;
   $temp_4=$temp2;$temp_4 =~ s/[0-9,a-f]{8}$//;
   $var1 = unpack('B*',pack('H*',$temp_1));
   $var2 = unpack('B*',pack('H*',$temp_2));
   $var3 = unpack('B*',pack('H*',$temp_3));
   $var4 = unpack('B*',pack('H*',$temp_4));
   print FH1 "$var1\n$var2\n$var3\n$var4\n"; }
close(FH);
close(FH1);
```

In this above lines of perl code the input file is passed to the program as an argument. This file is in hex and has four risc-v instructions in each line. This code reads each line separates them, then unpacks the hex code and packs them into binary risc-v instructions each in one line in a file “block.hex”.

```
$pc=1;
open(FH2, "<block.hex") or die "Couldn't open file file.txt, $!";
open(FH3, ">final.hex") or die "Couldn't open file file.txt, $!";
```

```
while($var = <FH2>)
{
   chomp($var);
   if($var =~ /1100011$/ || $var =~ /1100111$/ || $var =~ /1101111$/)
   {
      push(@comm,$pc);
      @abs=split('',$var);
      if($var=~/1101111$/)
      {
         @imm=@abs[0,0,0,0,0,0,0,0,0,0,0,0,0,12,13,14,15,16,17,18,19,11,1,2,3,4,5,6,7,8,9,10];
         $s_imm=join('',@imm);
         $offset=bin2dec($s_imm);
         if($offset != 0)
         {
               $lead = $offset/2 + $pc;
               push(@leaders,$lead);
         }
      }
      if($var=~/1100011$/)
      {
            @imm2=@abs[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,24,1,2,3,4,5,6,20,21,22,23];
            $s_imm2=join('',@imm2);
            $offset2=bin2dec($s_imm2);
            if($offset2 != 0)
            {
               $lead2=$offset2/2+$pc;
               push(@leaders,$lead2);
            }
      }
   }
   $pc=$pc+1;
}
sub bin2dec
{
    @as=@_;
    @ad=split('',$as[0]);
    if($ad[0] eq '1')
    {
      $ab=4294967296-unpack("N", pack("B32", substr("0" x 32 . shift, -32)));
      return -$ab;
   }
    else{return unpack("N", pack("B32", substr("0" x 32 . shift, -32)));}
}
```

The above lines of code read each instruction and identifies jump and branch instructions using the op-code. Once these instructions are identified, the target of the jump or branch needs to be calculated . This is done using the offset stored in the variable of the same name, calculated from the immediate part of these instructions stored in the array *@imm*. It is to be remembered that, the immediate encodes a signed offset in multiples of 2 bytes. But the pc variable is just a counter and the original program counter, hence only half the offset is added to correct the issue. The offset is sign-extended and added to the pc to form the jump target address. The offset is calculated in decimal using the bin2dec subroutine.

```
@leaders=sort(@leaders);
@leaders=unique(@leaders);
my %array2_elements;
@array2_elements{ @comm } = ();
@leaders = grep ! exists $array2_elements{$_}, @leaders;
@origleaders=@leaders;
close(FH2);

sub unique
{
    my %seen;
    grep !$seen{$_}++, @_;
}
```

The pc value pointing to each leader is stored in the leaders array. This array is sorted i-e the leader which is first executed is has lower index. This sorted array may have redundant entries, and such redundancy is removed using the "unique" sub routine.

```
open(FH4, "<block.hex") or die "Couldn't open file file.txt, $!";
$pc1=1;
while($var1 = <FH4>)
{
   chomp($var1);
   if($var1 =~ /1100011$/ || $var1 =~ /1100111$/ || $var1 =~ /1101111$/)
   {
      print FH3 "$var1\n\n";
   }
   elsif($pc1==$leaders[0])
```

```
    {
      shift(@leaders);
      print FH3 "****************\n$var1\n";#identifier for leaders of the basic
         block.
    }
    else
    {
       print FH3 "$var1\n";
    }
    $pc1 = $pc1+1;
  }
  close(FH3);
  close(FH4);
```

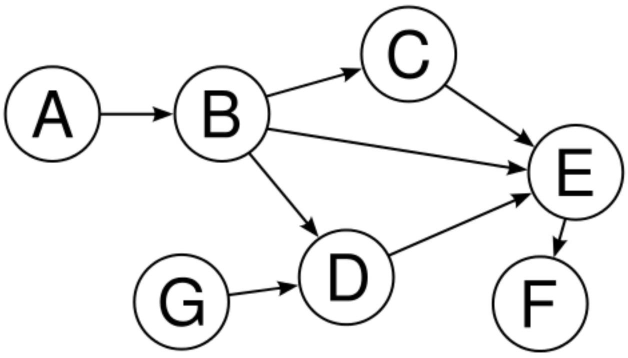
This part of the code takes the leaders and separates the basic blocks using identifiers so that the other programs can recognize them. The "final.hex" has the modified machine code with the basic blocks identified.

5.2 DAG Generation for each basic block :

In the second part, a dependency analysis is to be done on each basic block. Dependence analysis produces execution-order constraints between statements or instructions. Broadly speaking, a statement S2 depends on S1 if S1 must be executed before S2. If a computational problem can be divided into a number of subtasks, the data dependencies between these subtasks are usually described by means of a Directed Acyclic Graph (DAG).

A Directed Acyclic graph (DAG), is a directed graph with no directed cycles. That is, it is formed by a collection of vertices and directed edges, each edge connecting one vertex to another, such that there is no way to start at some vertex v and follow a sequence of edges that eventually loops back to v again A DAG allows multiple parents for each node. Both a tree and a DAG have a distinguished root. There are no cycles in the graph.

There are three possible data dependencies (RAR, RAW, WAW) that exist in the basic block. These dependencies provide the constraints for constructing the edges in the DAG. Each node in the DAG corresponds to an instruction in the basic block and each edge in the DAG corresponds to a data dependency between those two nodes. To check it there is an edge between any two nodes (instructions) the source and destination registers of the two instructions are checked to see if there is one of the three dependencies. Data dependency check is done on all the possible pair of nodes in the basic block to generate the DAG.



5.2.1 Implementation:

```
#! /usr/bin/python
def type_of_ins(temp):
   opcode = temp[-7:]
   if (opcode == "0110011") or (opcode == "0111011"):
      return temp[-20:-15],temp[-25:-20],temp[-12:-7]
   elif (opcode == "0000011") or (opcode == "0010011") or (opcode= "0011011") or
       (opcode= "1100111") :
      return temp[-20:-15],"",temp[-12:-7]
   elif (opcode= "1100011") or (opcode= "0100011"):
      return temp[-20:-15],temp[-25:-20],""
   elif (opcode= "1101111") or (opcode= "1110011") or (opcode= "0110111") or
       (opcode= "0010111"):
      return "","",temp[-12:-7]
   elif (opcode= "1010011"):
      if (opcode=="0101100") or (opcode=="1100000") or (opcode=="1110000") or
          (opcode=="1101000") or (opcode=="1111000"):
         return temp[-20:-15],"",temp[-12:-7]
      else:
         return temp[-20:-15],temp[-25:-20],temp[-12:-7]
   elif (opcode= "1110011"):
      if(temp[-15:-12]=="001"):
         return temp[-20:-15],"",temp[-12:-7]
      else:
         return "","",temp[-12:-7]
   elif (opcode= "0001111") or (opcode= "0000000"):
      return "","",""
   else:
        print "Error. Instruction:",temp
        sys.exit(1)
```

The above python function is defined to decode a RISC-V instruction that is given to it as an input. It decodes the instruction and determines the source registers and destination registers. There can be at most two source registers and one destination register. Absence any one of these registers, the function will return an empty string. The purpose for this function to help detect the dependencies in a basic block.

```
os.system("./basicblocks.pl <path to input hex code>")
f1= open("final.hex","r")
l=f1.readlines()
l=l[2048:]
lll = []
w=0
lll.append([w])

for i in l:
   if (i=='\n' or i=="****************\n"):
      w=w+1
      lll.append([w])
   else:
      lll[w].append(i)

f1.close()
```

The basic block script is called upon to generate basic blocks and the output is stored in “final.hex” . The first 2048 instructions in the generated file are null instructions. During simulation of the rocket-chip the pc is set to start from 0X2000 , hence the first 512 lines of the input file are always zero’s . Since these are of no consequence, these are omitted and rest are stored in the ’l’ variable.

Then using the identifiers, basic block are stored in the variable ’lll’ which is a list of lists in python. Each element of this list corresponds to a basic block. The first elements of these sub-lists are the index numbers of the corresponding basic blocks and all the remaining elements are the instructions present in those basic blocks.

```
for q in range(w+1):
```

```python
d = []
for z in range(q):
   for c in lll[z][1:]: d.append(c)
l = lll[q][1:-1]
k = len(l)
for j in range(k):
   l[j]=l[j].strip()
n=0
dag=[]
while n<k:
   dag.append([n])
   n=n+1
for i in range(k):
   temp1 = l[i]
   src1a,src2a,desa = type_of_ins(temp1)
   for j in range(i+1,k):
      temp2 = l[j]
      src1b,src2b,desb = type_of_ins(temp2)
      if ((desa==src1b or desa==src2b) and desa is not "") or ((desb==src1a or
          desb==src2a) and desb is not "") or (desa==desb and desa is not ""):
         dag[i].append(j)
```

Each iteration of the main for loop corresponds to optimization of each basic block. The last instruction of the basic block is either a jump or a branch statement, and hence it would not be rearranged.

All the instructions that are to be rearranged, are stored in 'l'. The DAG to be generated is stored in list variable 'dag'. The dag variable is constructed in the following manner:

- Each element is a list
- All the elements other than the first element in that sub-list corresponds to the adjacency list of the first element of that list i-e there exists an edge between the first element and each of the rest of the elements.
- The edge that exists is directed from the first element to the other elements.

The code takes all possible pairs of instruction in a basic block and checks for dependencies using the 'type_of_ins' function and its output. The DAG is updated whenever such dependency is detected. This dag is then further used for generating the possible topological sorts.

5.3 Level order sorting of DAG :

Every directed acyclic graph has a topological ordering, an ordering of the vertices such that the starting node of every edge occurs earlier in the ordering than the ending node of the edge. In general, this ordering is not unique, a DAG has a unique topological ordering if and only if it has a directed path containing all the vertices, in which case the ordering is the same as the order in which the vertices appear in the path. So any two graphs representing the same partial order have the same set of topological orders.

In the third part, the DAG that is generated is topologically sorted. A level order sorting is performed on the DAG of each basic block one at a time. At each level there can be many nodes. Suppose there are 'n' nodes at a particular level, then there can be n! different rearrangements, each having the same functionality. It is to be noted that not all possible topological sorts are used. Simulating all the possible arrangements on the target processor is too complex . Hence the aim would be to generate a better solution than the best.

So we would reorder the DAG level wise, one level at a time .One of such arrangement corresponds to least number of toggles , i-e the least amount of dynamic power consumption.That order would be retained as the algorithm traverses through the DAG.

All the different rearrangements in that level of the basic block are simulated keeping the other basic blocks intact and the number of toggles (dynamic power) consumed in each case can be observed through Value Change Dump (VCD). Value change dump (VCD) is an ASCII-based format for dump files generated by EDA logic simulation tools. The VCD file comprises a header section with date, simulator, and timescale information, a variable definition section, and a value change section. The number of lines in value change section in the VCD file determines the number of toggles in that particular simulation.Hence the arrangement with the least number of lines the VCD file would determine the optimal solution.

One thing to note is before simulating each rearrangement, the format of the input that is to be given to Rocket core should be changed to original format i.e. four instructions per line in hex. The rearrangement with least power is the best case and we fix this order in this level. Then the same process is repeated for all the further levels. Once all the levels are done, the best power optimized code for the first basic block is generated and the original code of this basic block is replaced with this code. Then the same process is repeated for all the other basic blocks as well. When the last basic block is done, the best power optimized code for all the basic blocks are got. Since all the initial basic block codes are replaced with these power optimized codes, we get the final power optimized code for the given input hex code. The second part and the third part of the algorithm is written in python.

5.3.1 Implementation:

```
n=0
X = []
Y = []
bp = []
while(len(Y)<k):
   X = []
   for i in range(k):
      temp = 0
      for j in dag:
         if ((dag.index(j) != i) and (i in j) and (dag.index(j) not in Y)):
            temp = 1
            break
      if (temp==0 and (i not in Y)):
         X.append(i)

   permutations_of_X = list(itertools.permutations(X))
   Y = Y + X
   cnt = 0
   vcd_list = []
   jobs = []
   bcd_list = Queue()
   for i in permutations_of_X:
      f2= open("input.hex","w")
      for c in d: f2.writelines(c)
      for wr in bp:
         f2.write(l[wr]+'\n')
      for p in i:
         f2.write(l[p]+'\n')
      for j in range(k):
```

```
    if(j not in Y):
        f2.write(l[j]+'\n')
f2.write(lll[q][-1])
for v in range(q+1,w+1):
    for e in lll[v][1:]: f2.writelines(e)
f2.close()
#Reformatting to four instructions in hex per line
fr = open("input.hex","r")
fw = open("inputv2"+str(cnt)+".hex","w")
l2=fr.readlines()
k2 = len(l2)
i2 = 0
for lol in range(512):
    fw.write("00000000000000000000000000000000\n")
for line2 in l2:
    line2 = line2.strip()
    if(i2==0):
        x2 = line2
        i2 = 1
    elif(i2==1):
        x3 = line2
        i2 = 2
    elif(i2==2):
        x4 = line2
        i2 = 3
    elif(i2==3):
        x5 = line2
        fw.write("{0:0>8x}".format(int(x5, 2)) + "{0:0>8x}".format(int(x4, 2))
             + "{0:0>8x}".format(int(x3, 2)) + "{0:0>8x}".format(int(x2, 2)) +
             '\n')
        i2 = 0
if(i2==1):
    fw.write("{0:0>8x}".format(int("0", 2)) + "{0:0>8x}".format(int("0", 2))
         + "{0:0>8x}".format(int("0", 2)) + "{0:0>8x}".format(int(x2, 2)) +
         '\n')
if(i2==2):
    fw.write("{0:0>8x}".format(int("0", 2)) + "{0:0>8x}".format(int("0", 2))
         + "{0:0>8x}".format(int(x3, 2)) + "{0:0>8x}".format(int(x2, 2)) + '\n')
if(i2==3):
    fw.write("{0:0>8x}".format(int("0", 2)) + "{0:0>8x}".format(int(x4, 2)) +
         "{0:0>8x}".format(int(x3, 2)) + "{0:0>8x}".format(int(x2, 2)) + '\n')
fw.close()
fr.close()
```

```
        #Reformatting done
        if(cnt == 9):
            p = Process(target=run,args=(cnt,))
            jobs.append(p)
            cnt = -1
            for xy in jobs:
                xy.start()
            #wait for prev 10 jobs to be done
            for xy in jobs:
                xy.join()
            tcd = [bcd_list.get() for xy in jobs]
            vcd_list= vcd_list + tcd
            print "tcd_list = ",vcd_list
            jobs = []
            bcd_list = Queue()
        else:
            p = Process(target=run,args=(cnt,))
            jobs.append(p)
        cnt = cnt+1
        xyz = xyz+1
    if cnt!=10:
        for xy in jobs:
            xy.start()
        for xy in jobs:
            xy.join()
        tcd = [bcd_list.get() for xy in jobs]
        vcd_list= vcd_list + tcd
    abc = 0
    lowest = vcd_list[0]
    l_i = 0
    for count in vcd_list:
        if(count<lowest):
            lowest = count
            l_i = abc
        abc = abc + 1
    #bp is updated
    for stuff in permutations_of_X[l_i]:
        bp.append(stuff)
bbp = []
g = 0; fl=1
for u in bp:
    bbp.append(lll[q][fl+u])
    g = g+1
```

```
  for ind in range(g): lll[q][fl+ind] = bbp[ind]
def run(count):
    os.system("./simv-DefaultVLSIConfig-debug15 -q +ntb_random_seed_automatic
       +dramsim +verbose +vcdfile=ss"+str(count)+".vcd +max-cycles=100000000
       +loadmem=inputv2"+str(count)+".hex 2> sss"+str(count)+".out")
    x = sum(1 for line in open("ss"+str(count)+".vcd"))
    bcd_list.put(x)
```

In the code above, X is the list that has all the instructions which are in a level that is going be optimized in this iteration. Once a level is optimized the outward edges from instructions at this level are not considered as inward edges for instructions of further levels. So the instructions in X don't have any inward edges. To compute X, a loop is used which iterates over all instructions in a basic block. Each instruction in the basic block is checked for any inward edges, and those with none are added to X .

The "Y" is a list that contains the instructions in the levels that have been optimized. After X is computed, the simulations corresponding to all possible rearrangements in the current level are executed. All the instructions of the preceding basic blocks are updated in list "d". Similarly all the instructions of the preceding levels in the current basic block are updated in list "bp".

So, to run a simulation, firstly all the instructions in "d" and then the instructions in "bp" are written into a file. The rearrangement of the instructions in current level, the remaining instructions in the current basic block, and the instructions of successive basic blocks are written without change.

The format is changed from one instruction in binary per line to four instructions in hex per line and can now be run using simulator. All the different rearrangements of a current level are simulated and the VCD files of each simulation are read. The VCD file is used to calculate the number of toggles. The one with least number of toggles is retained at that level for further simulations. The variable "bp" is updated with this best possible rearrangement of the current level.

The same process is repeated for all the levels iteratively. All the levels are done when the length of list "Y" becomes equal to "k" i.e. length of basic block. At that time "bp" has all the optimum rearrangements at all the levels i.e. best possible order of instructions for the current basic block . After the completion of a basic block, the original order of instructions in that basic block in list “lll” is replaced by optimized order of instructions.

5.4 Parallel Processing:

The python code also includes multi-threading and parallel processing. There are multiple simulations that need to be executed, where each simulation is independent of the other. Hence there is potential for parallelization. It is time consuming to get the final power optimized code for large inputs of hex code. So, for that reason, using multi-threading constructs in python to execute simulations in parallel which reduces the time taken to a great extent. Multiprocessing package available in python can be used to implement this.

Depending on the application, two common approaches in parallel programming are either to run code via threads or multiple processes, respectively. Once "jobs" are submitted to different threads, those jobs can be pictured as "subtasks" of a single process and those threads will usually have access to the same memory areas (i.e., shared memory). This approach can easily lead to conflicts in case of improper synchronization, for example, if processes are writing to the same memory location at the same time.A safer approach (although it comes with an additional overhead due to the communication overhead between separate processes) is to submit multiple processes to completely separate memory locations (i.e., distributed memory): Every process will run completely independent from each other.

Multiprocessing is a package that supports spawning processes using an API similar to the threading module. The multiprocessing package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using subprocesses instead of threads. Due to this, the multiprocessing module allows the programmer to fully leverage multiple processors on a given machine.

The above code simulates 10 simulations in parallel. The variable ’cnt’ counts from 0 to 9, as jobs are added to start. Once count becomes 9 all the jobs are started. The code waits till all the 10 jobs are completed. Each job takes up a separate thread. The results of all the ten simulations are queued and are stored in ’bcd_list’ and finally added to ’vcd_list’. Once all the permutations in a particular level are completed, the least number of toggles is ’min(vcd_list)’.

The run function is called for each permutation which calls the command for simulation.

6 Output file:

6.1 Verification of functionality:

The final optimized code should have the same functionality as the original code. The integrity of the code should not suffer due to the rearrangements. While simulating the Rocket core, an output file and VCD file are generated. The output file contains a cycle-by-cycle dump of write-back stage of the pipeline.

To verify this, the destination registers of each instruction, updated at the write-back stage are dumped into the output file. The output files of both the optimized code and the original code are compared to verify the functionality. The order of the destination register in both the files need not be the same. Hence both the files are sorted, and compared line by line to verify the functionality. It is to be noted that only valid write back stages are to be considered, that is whenever the processor encounters a branch or a jump statement, all the instructions inside the pipeline are cleared, hence such instructions executed in the write back stage are not valid. To clearly distinguish the valid write back stage, register value is only dumped when the write back is valid indicated by a boolean check bit.

```
os.system("sort original.out -o file1.out && sort optimized.out -o file2.out && comm
    -3 file1.out file2.out >>file3.out")
difference = sum(1 for line in open('file3.out'))
if(difference>0):
   print "difference = ",difference
    sys.exit(1)
```

A sample of the output file :

```
W[r  2=0000000006db6db7][1]
W[r  2=0000006db6db7000][1]
W[r  1=0000000000008000][1]
W[r  2=0000000006db7000][1]
W[r  2=0000006db6db6db7][1]
W[r  3=6db6db6db6dbebb7][1]
W[r  2=0006db6db6db7000][1]
W[r  2=0006db6db6db6db7][1]
```

```
W[r  2=6db6db6db6db7000][1]
W[r  1=0000000000007e00][1]
W[r  3=0000000000001200][1]
```

Sorting it would result in :

```
W[r  1=0000000000007e00][1]
W[r  1=0000000000008000][1]
W[r  2=0000000006db6db7][1]
W[r  2=0000000006db7000][1]
W[r  2=0000006db6db6db7][1]
W[r  2=0000006db6db7000][1]
W[r  2=0006db6db6db6db7][1]
W[r  2=0006db6db6db7000][1]
W[r  2=6db6db6db6db7000][1]
W[r  3=0000000000001200][1]
W[r  3=6db6db6db6dbebb7][1]
```

6.2 Resolving JALR instruction:

In the indirect jump instruction JALR (jump and link register) the target address is obtained by adding the 12-bit signed I-immediate to the register rs1, then setting the least-significant bit of the result to zero. The address of the instruction following the jump (pc+4) is written to register rd.

Hence unless and until all the instructions preceding the JALR instruction are executed the target address cannot be resolved. Since the target address need not be evaluated at real time, this issue can be solved by running the simulation and using the output file.

The write back of each instruction along with the instruction , pc value, source registers are dumped into the output file. Using the value of rs1 at each of these JALR instruction, the target address is evaluated. These target addresses are now treated as leaders of a basic blocks and so, the the issue is resolved.

```
C0:   263  [1]  pc=[00000002068] W[r  2=6db6db6db6db6db7][1]  R[r  2=6db6db6db6db7000]
        R[r23=0000000000001200]  inst=[db710113] DASM(db710113)
C0:   264  [1]  pc=[0000000206c] W[r  3=6db6db6db6dbed77][1]  R[r  1=0000000000007fc0]
        R[r  2=6db6db6db6db6db7]  inst=[022081b3] DASM(022081b3)
C0:   265  [1]  pc=[00000002070] W[r29=0000000000001000][1]  R[r  0=0000000000000000]
        R[r  0=0000000000000000]  inst=[00001eb7] DASM(00001eb7)
C0:   266  [1]  pc=[00000002074] W[r29=0000000000001240][1]  R[r29=0000000000001000]
        R[r  0=0000000000000000]  inst=[240e8e9b] DASM(240e8e9b)
```

```
C0:   267  [1]  pc=[00000002078] W[r28=0000000000000021][1]  R[r  0=0000000000000000]
      R[r  1=0000000000001200]  inst=[02100e13]  DASM(02100e13)
C0:   268  [0]  pc=[00000002078] W[r  3=0000000000001240][1]  R[r  0=0000000000000000]
      R[r  1=0000000000001200]  inst=[02100e13]  DASM(02100e13)
C0:   269  [0]  pc=[00000002078] W[r  0=0000000000000021][0]  R[r  0=0000000000000000]
      R[r  1=0000000000001200]  inst=[02100e13]  DASM(02100e13)
C0:   270  [0]  pc=[00000002078] W[r  0=0000000000000021][0]  R[r  0=0000000000000000]
      R[r  1=0000000000001200]  inst=[02100e13]  DASM(02100e13)
C0:   271  [1]  pc=[0000000207c] W[r  0=0000000000000000][0]  R[r  3=0000000000001240]
      R[r29=0000000000001240]  inst=[45d19c63]  DASM(45d19c63)
```

7 Results of RTL Simulations :

S.No	Toggles before optimization	Toggles after optimization	% decrease in no. of toggles
1.	1171827	1159129	1.08
2.	1218765	1197846	1.75
3.	1140663	1126999	1.20
4.	1135007	1122262	1.12
5.	1346098	1317770	2.10
6.	1113520	1099355	1.27
7.	1208503	1197454	0.90
8.	1143237	1121679	1.88
9.	1135629	1123117	1.10
10.	1135610	1122451	1.16

8 Need for Post- Synthesis Simulation:

The number of toggles observed through VCD file is not an accurate way of depicting the power consumption. For instance if there is a register of 8 bit width and its value changes from 8'h00 to 8'hff then number of toggles is 8 but VCD indicates only a change in value but not the exact number of toggles.

So, to get the exact number of toggles, the gate level Netlist needs to be synthesized from the RTL of the processor. Netlist describes the interconnections between different modules with the hierarchy reduced to the transistor level. Every interconnect, delay, leakage power and capacitances are modeled for the design in the Netlist generated. Hence the accurate Power consumption in watts for executing the code on the processor can be found from the VCD generated by the post synthesis simulation, using tools such as Prime Time. Hence more accurate results can only be achieved by rearrangements guided by post synthesis simulation. During the post synthesis simulations, the number of simulation time would increase and hence multiple simulations would not be possible for an entire basic block. The best way to tackle this problem would be to execute two instructions in a simulation and assign the number of toggles between them as weights to the edges. Now the problem would be to construct a Hamiltonian path with the least cost.

8.1 Hamiltonian Path and the Traveling Salesman Problem:

8.1.1 Hamiltonian Path:

In graph theory, a Hamiltonian path is a path in an undirected or directed graph that passes through each node exactly once. A Hamiltonian cycle is a Hamiltonian path that is a cycle. Determining whether such paths and cycles exist in graphs is the Hamiltonian path problem, which is NP-complete. For directed graphs, where each edge of a path or cycle can only be traced in a single direction i.e., the vertices are connected with arrows and the edges traced "tail-to-head".

Now consider in a particular level in a basic block there are three instructions I_1, $I_2$and I_3. Let there be $(n_1, n_2), (n_3, n_4)$and (n_5, n_6) toggles between each instruction found by the post synthesis simulation of each pair of instructions in both directions. Then the best possible arrangement is a Hamiltonian path with the least cost associated with it. This solution boils down to a Traveling Salesman Problem.

Any problem that shares these common elements:

- *a traveler*
- *a set of sites*
- *a cost function for travel between pairs of sites*
- *a need to tour all the sites*
- *a desire to minimize the total cost of the tour*

is known as a traveling salesman problem, or TSP.

There are two strategies for solving traveling salesman problem:

- Exhaustive Search: Make a list of all possible Hamilton circuits. For each circuit in the list, calculate the weight of the circuit. From all the circuits, choose a circuit with least total weight. This is your optimal tour.
- Nearest Neighbour solution: Start from the home city. From there go to the city that is the cheapest to get to. From each new city go to the next new city that is cheapest to get to. When there are no more new cities to go to, go back home.

The negative aspect of the brute-force algorithm is the amount of effort that goes into implementing the algorithm, which is (roughly) proportional to the number of Hamilton circuits in the graph.The brute-force, exhaustive search algorithm is an algorithm for which the number of steps needed to carry it out grows disproportionately with the size of the problem. The trouble with this algorithm is that it can realistically be carried out only when the problem is small.

In the second heuristic the amount of computational effort required to implement the algorithm grows in some reasonable proportion with the size of the input to the problem. The main problem with this algorithm is that it is not an optimal algorithm. The relative error would decide the best approach.

9 Conclusion:

The RTL simulation results shows positive results. The full extent of power that can be optimized, can be realized through the post synthesis evaluation. This report discusses a novel idea with potential for further study and analysis.

References

[1] http://riscv.org/

[2] http://riscv.org/download.html#tab_rocket

[3] http://riscv.org/download.html#tab_isaspec

[4] http://www-inst.eecs.berkeley.edu/~cs250/fa13/