Design Concepts with System Architect: Power & Hazard Modeling

John Leidel

Chief Scientist, Tactical Computing Laboratories

ver 2022.09.13

Tutorial Series

- Level 0: Introduction to System Architect
- Level 1: System Architect Design Concepts and Developing a basic RISC processor
- Level 2: Instruction-Level (StoneCutter) Implementation Concepts
- Level 3: Advanced Design Concepts
- Level 4: System Architect Plugins and Integrating External RTL
- **Power and Hazard Modeling with DHDT & POAR**

Overview

- Power/Hazard Modeling Tools Overview
- POAR Power/Area Modeling
- DHDT Power/Hazard Modeling

Power/Hazard Modeling Tools Overview

System Architect Rapid Design Evaluation Tooling

System Architect Power/Hazard Modeling Tools

POAR

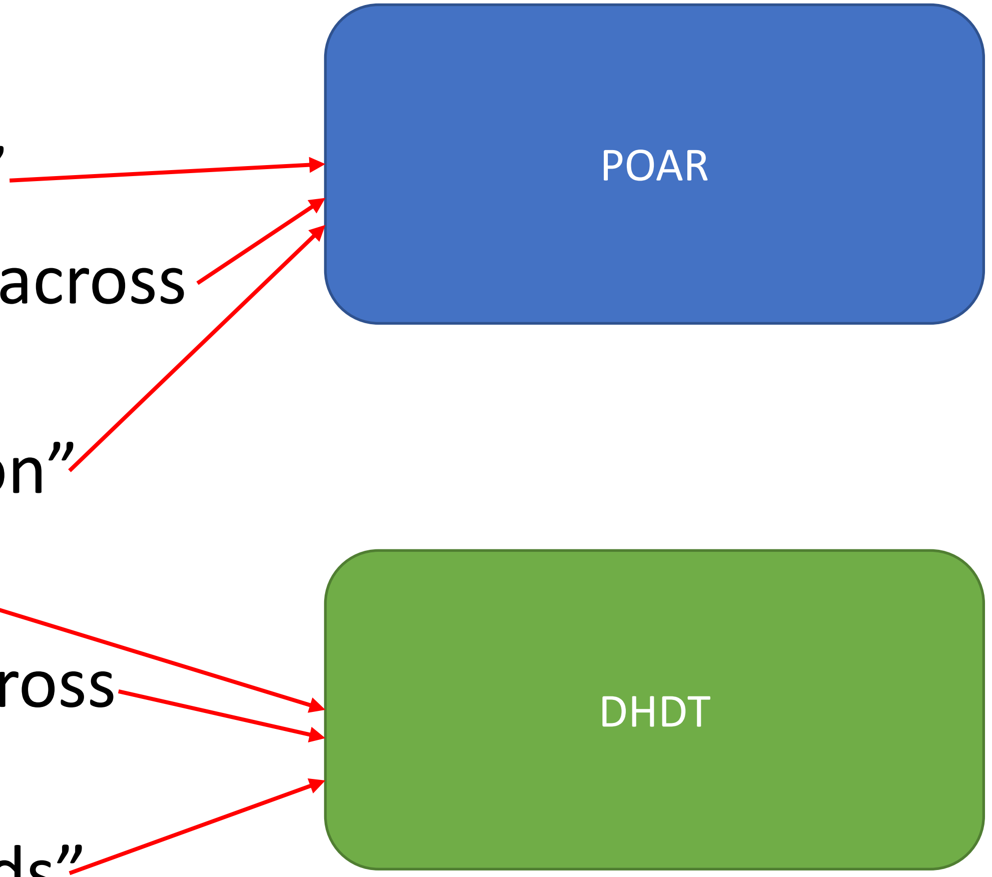
- *<u>Coarse grained</u>* power and area modeling tool
- Utilizes the CoreGen YAML design inputs as the basis for the power/area modeling
- Permits user-defined power/area values
  - External configuration files that outline power/area for specific hardware units
- Permits a user-defined constant routing overhead
  - Process-specific routing overhead
- Variety of output formats for further processing/analysis

DHDT

- *<u>Fine grained</u>* power and hazard modeling tool
- Utilizes the CoreGen YAML and the StoneCutter LLVM IR as the basis for the power/hazard modeling
- Constructs a graph of all the candidate signals & clocked pipeline stages for each instruction definition/VLIW stage
- Permits user-defined power value inputs
- Accepts assembly, binary and hex inputs to track power/hazards for individual instructions with accumulated totals
- CSV, ASCI output as well as DOT graph outputs for the internal graph structures

What do I use and when?

- “I want to see basic power/area models”
- “I want to compare basic power models across designs”
- “I want to see basic design area utilization”
- “I want to see per instruction power”
- “I want to see detailed power models across different process nodes”
- ”I want to model VLIW instruction hazards”



Caveats?

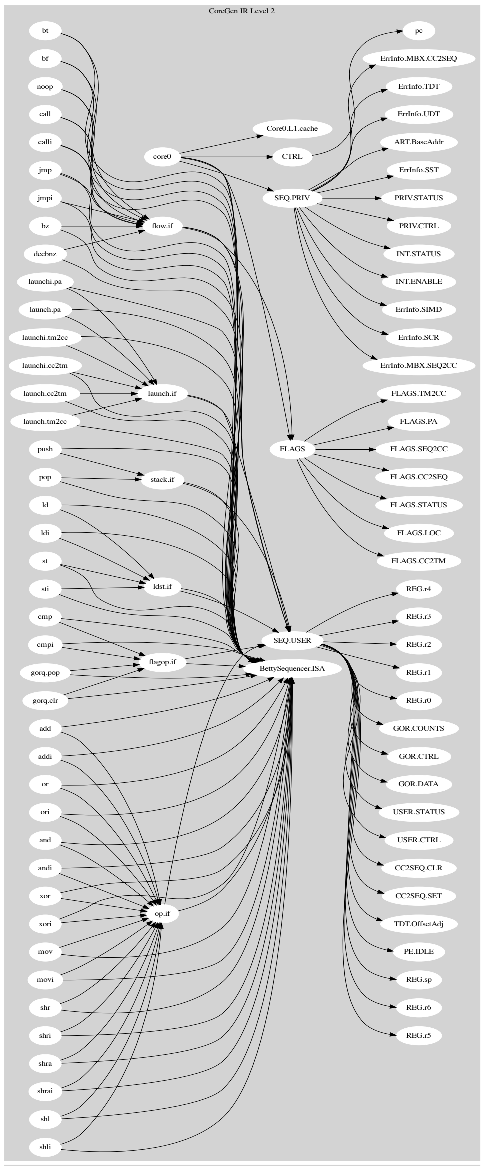
- POAR and DHDT are both approximations
    - They do not consider switching/transient power
    - They have a finite notion of routing congestion (tool dependent)
    - They rely upon basic power/area model inputs that are process specific
    - *YOUR* model inputs are *YOUR* responsibility
- We seek to maintain tool performance without sacrificing functionality
- If you would like new features, submit a Github Issue!

POAR Power/Area Modeling

Coarse Grained Power/Area Modeling

POAR [Power Area Model]

- Utilizes the CoreGen design graph input (YAML) to build a graph of included hardware modules
  - Optionally also uses the signal maps generated by the StoneCutter tool
- For all known design nodes, accumulates total power and total area
  - Approximate model of power: TDP
  - Approximate model of area: “gates”
- Outputs power and area per node type as well as accumulated totals



POAR Info Options

- --help : Prints the help menu
- --version : Prints the version info

```
$> poar --help
$> poar --version
```

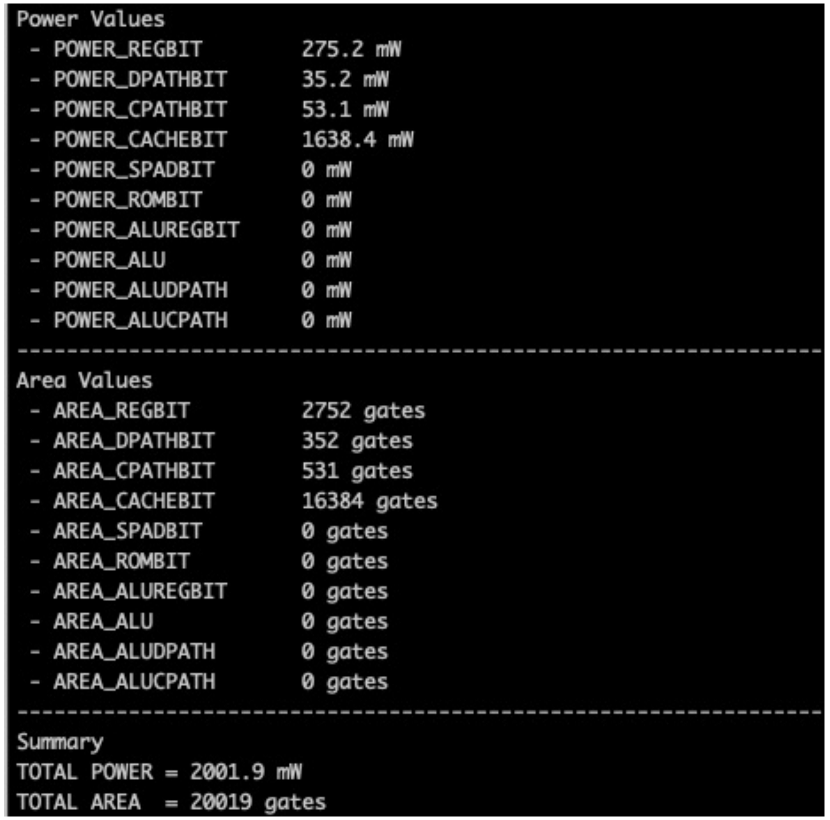
POAR Runtime Options

- Required:
  - --design /path/to/CoreGen.yaml : design input
- Optional:
  - --config /path/to/config.yaml : Sets the power/area node configuration
  - -- sigmap /path/to/sigmap.yaml : Stonecutter-generated signal map
  - --overhead NN : Sets the constant routing/wire overhead for area calculations

```
$> poar --design test.yaml
$> poar --design test.yaml --sigmap test_sigmap.yaml
$> poar --design test.yaml --overhead 8.1
```

POAR Output Options

- Default output is ASCII text printed to the console
- Additional output options:
  - --text : ASCII text output
  - --yaml : YAML output
  - --latex : LaTeX output
  - -- xml : XML output

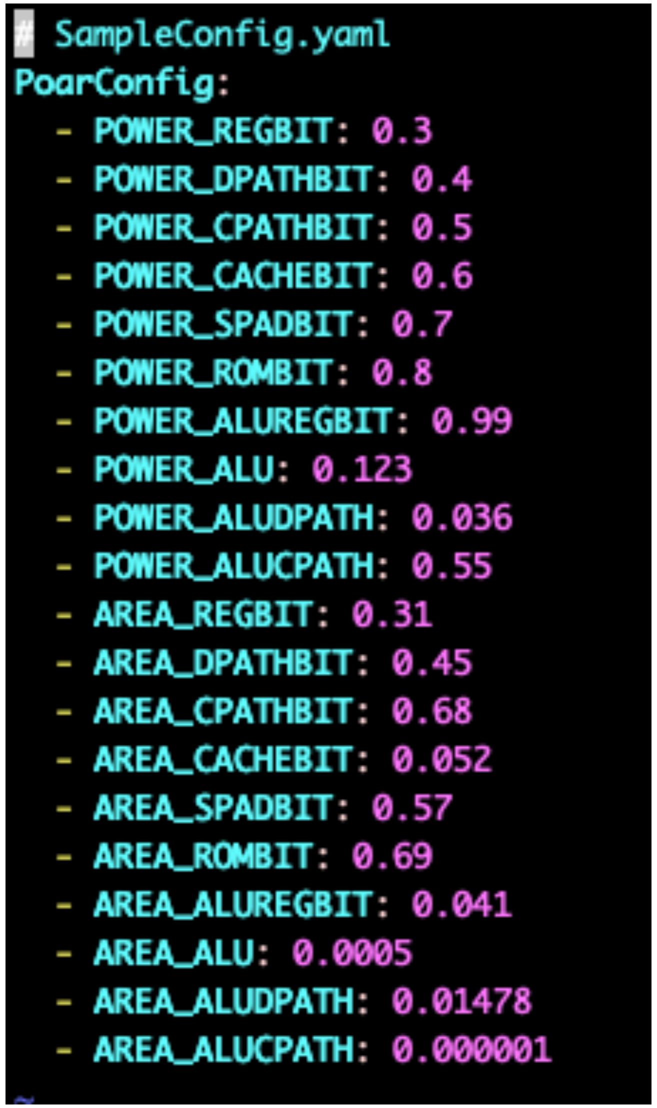


```
Power Values
 - POWER_REGBIT         275.2 mW
 - POWER_DPATHBIT       35.2 mW
 - POWER_CPATHBIT       53.1 mW
 - POWER_CACHEBIT       1638.4 mW
 - POWER_SPADBIT        0 mW
 - POWER_ROMBIT         0 mW
 - POWER_ALUREGBIT      0 mW
 - POWER_ALU            0 mW
 - POWER_ALUDPATH       0 mW
 - POWER_ALUCPATH       0 mW
----------------------------------------------------------------
Area Values
 - AREA_REGBIT          2752 gates
 - AREA_DPATHBIT        352 gates
 - AREA_CPATHBIT        531 gates
 - AREA_CACHEBIT        16384 gates
 - AREA_SPADBIT         0 gates
 - AREA_ROMBIT          0 gates
 - AREA_ALUREGBIT       0 gates
 - AREA_ALU             0 gates
 - AREA_ALUDPATH        0 gates
 - AREA_ALUCPATH        0 gates
----------------------------------------------------------------
Summary
TOTAL POWER = 2001.9 mW
TOTAL AREA  = 20019 gates
```

```
$> poar --design test.yaml --text
$> poar --design test.yaml --yaml
$> poar --design test.yaml --latex
$> poar --design test.yaml --xml
```

POAR Configuration Options

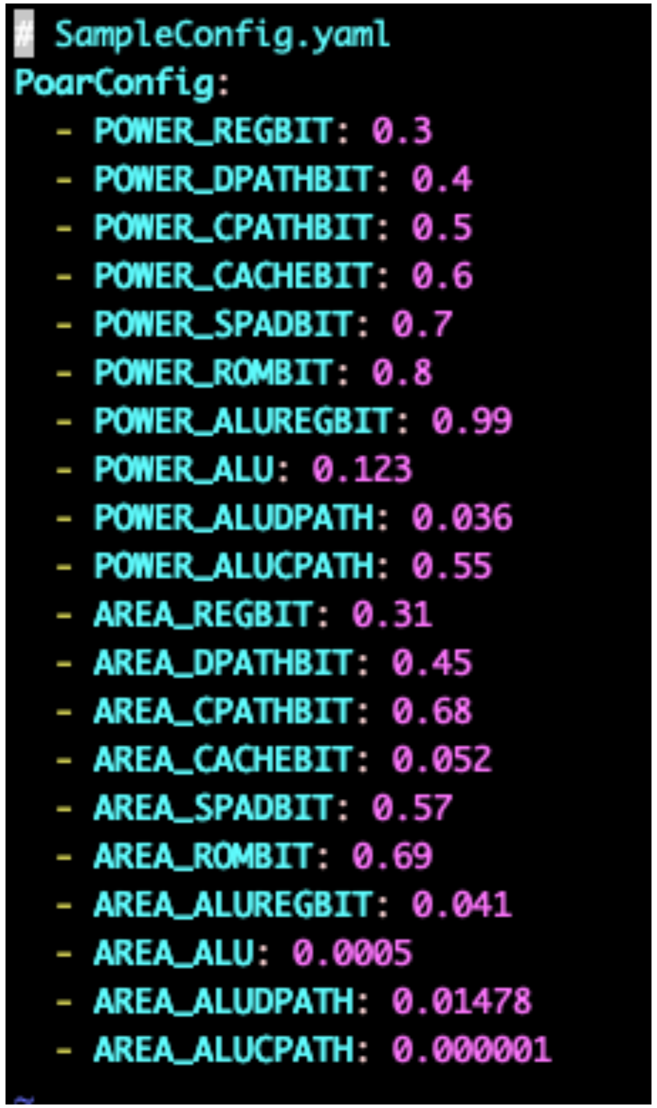
- ***POWER_REGIBIT***: Power per bit in a register
- ***POWER_DPATHBIT***: Power per data path bit
- ***POWER_CPATHBIT***: Power per control path bit
- ***POWER_CACHEBIT***: Power per bit in the cache
- ***POWER_SPADBIT***: Power per bit in the scratchpad
- ***POWER_ROMBIT***: Power per bit in the internal boot/uCode ROM
- ***POWER_ALUREGBIT***: Power per bit in an ALU register
- ***POWER_ALU***: Power per ALU bit
- ***POWER_ALUDPATH***: Power per ALU data path
- ***POWER_ALUCPATH***: Power per ALU control path



```
# SampleConfig.yaml
PoarConfig:
  - POWER_REGBIT: 0.3
  - POWER_DPATHBIT: 0.4
  - POWER_CPATHBIT: 0.5
  - POWER_CACHEBIT: 0.6
  - POWER_SPADBIT: 0.7
  - POWER_ROMBIT: 0.8
  - POWER_ALUREGBIT: 0.99
  - POWER_ALU: 0.123
  - POWER_ALUDPATH: 0.036
  - POWER_ALUCPATH: 0.55
  - AREA_REGBIT: 0.31
  - AREA_DPATHBIT: 0.45
  - AREA_CPATHBIT: 0.68
  - AREA_CACHEBIT: 0.052
  - AREA_SPADBIT: 0.57
  - AREA_ROMBIT: 0.69
  - AREA_ALUREGBIT: 0.041
  - AREA_ALU: 0.0005
  - AREA_ALUDPATH: 0.01478
  - AREA_ALUCPATH: 0.000001
```

POAR Configuration Options cont.

- ***AREA_REGIBIT***: Area per bit in a register
- ***AREA_DPATHBIT***: Area per data path bit
- ***AREA_CPATHBIT***: Area per control path bit
- ***AREA_CACHEBIT***: Area per bit in the cache
- ***AREA_SPADBIT***: Area per bit in the scratchpad
- ***AREA_ROMBIT***: Area per bit in the internal boot/uCode ROM
- ***AREA_ALUREGBIT***: Area per bit in an ALU register
- ***AREA_ALU***: Area per ALU bit
- ***AREA_ALUDPATH***: Area per ALU data path
- ***AREA_ALUCPATH***: Area per ALU control path



```
# SampleConfig.yaml
PoarConfig:
  - POWER_REGBIT: 0.3
  - POWER_DPATHBIT: 0.4
  - POWER_CPATHBIT: 0.5
  - POWER_CACHEBIT: 0.6
  - POWER_SPADBIT: 0.7
  - POWER_ROMBIT: 0.8
  - POWER_ALUREGBIT: 0.99
  - POWER_ALU: 0.123
  - POWER_ALUDPATH: 0.036
  - POWER_ALUCPATH: 0.55
  - AREA_REGBIT: 0.31
  - AREA_DPATHBIT: 0.45
  - AREA_CPATHBIT: 0.68
  - AREA_CACHEBIT: 0.052
  - AREA_SPADBIT: 0.57
  - AREA_ROMBIT: 0.69
  - AREA_ALUREGBIT: 0.041
  - AREA_ALU: 0.0005
  - AREA_ALUDPATH: 0.01478
  - AREA_ALUCPATH: 0.000001
~
```

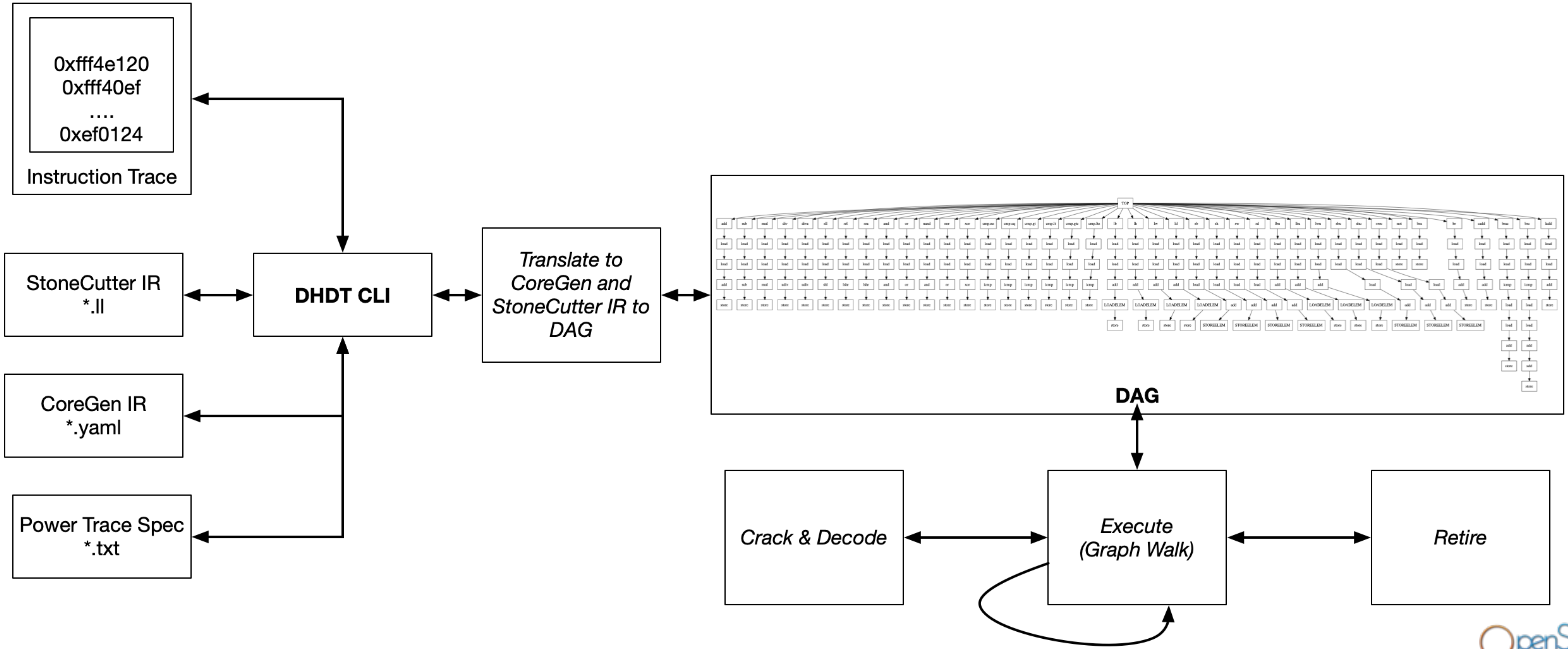
DHDT Power/Hazard Modeling

Fine Grained Power/Hazard Modeling

DHDT [Deep Hardware Design Tool]

- Fine grain power and hazard modeling tool
  - Fine grain in the scope of System Architect. NOT equivalent to a SPICE model
- Utilizes the CoreGen YAML design input and the StoneCutter compiler IR to build an internal graph representation of the target device
- Models individual instructions as they execute (traverse the graph) and accumulate per-instruction power & hazarding
- Allows users to investigate how individual instructions behave with respect to power utilization (or groups of instructions)
  - Requires the user to input instruction sequences
- Outputs data in ASCII and CSV

DHDT [Deep Hardware Design Tool]

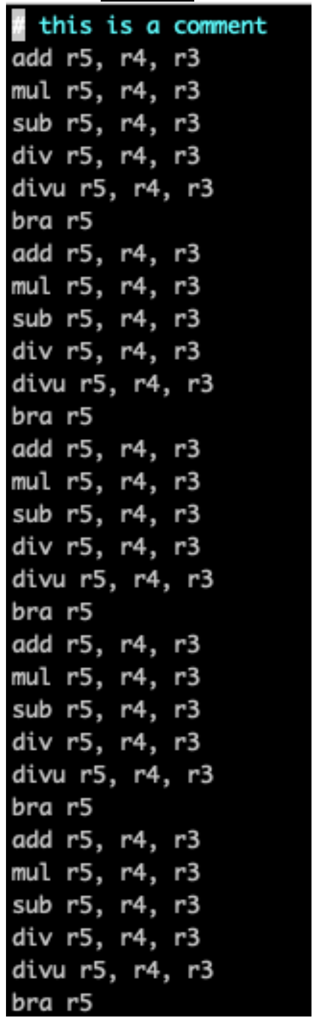


DHDT [Deep Hardware Design Tool]

- DHDT accepts “templates” that describe power values for various hardware subcomponents
  - Much like POAR, the power values are relative to a process node
  - DHDT power configuration templates contain much more detailed information
- Instruction trace inputs are in three forms (one instruction payload per line):
  - Assembly: Assembly mnemonics as defined by the CoreGen YAML
  - Binary: Binary encoded instruction values prefixed with *0b…*
  - Hex: Hex encoded instruction values prefixed with *0x…*

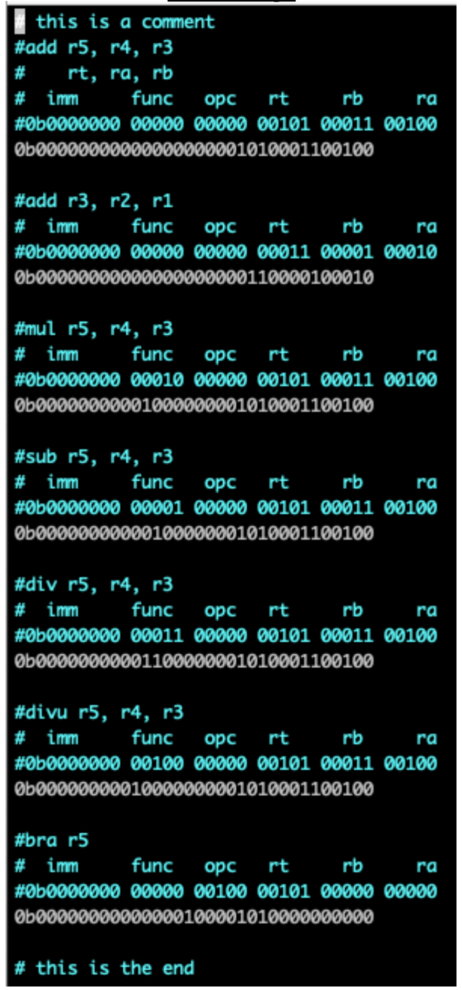
DHDT Instruction Trace Inputs

ASM



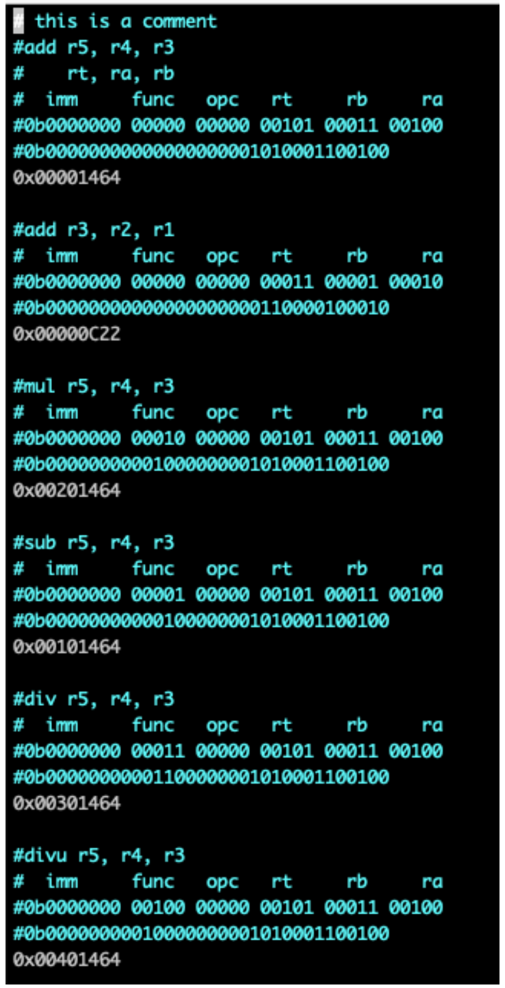
```
# this is a comment
add r5, r4, r3
mul r5, r4, r3
sub r5, r4, r3
div r5, r4, r3
divu r5, r4, r3
bra r5
add r5, r4, r3
mul r5, r4, r3
sub r5, r4, r3
div r5, r4, r3
divu r5, r4, r3
bra r5
add r5, r4, r3
mul r5, r4, r3
sub r5, r4, r3
div r5, r4, r3
divu r5, r4, r3
bra r5
add r5, r4, r3
mul r5, r4, r3
sub r5, r4, r3
div r5, r4, r3
divu r5, r4, r3
bra r5
add r5, r4, r3
mul r5, r4, r3
sub r5, r4, r3
div r5, r4, r3
divu r5, r4, r3
bra r5
```

Binary



```
# this is a comment
#add r5, r4, r3
#    rt, ra, rb
#  imm      func   opc    rt     rb     ra
#0b0000000 00000 00000 00101 00011 00100
0b00000000000000000001010001100100

#add r3, r2, r1
#  imm      func   opc    rt     rb     ra
#0b0000000 00000 00000 00011 00001 00010
0b00000000000000000000110000100010

#mul r5, r4, r3
#  imm      func   opc    rt     rb     ra
#0b0000000 00010 00000 00101 00011 00100
0b00000000001000000001010001100100

#sub r5, r4, r3
#  imm      func   opc    rt     rb     ra
#0b0000000 00001 00000 00101 00011 00100
0b00000000000100000001010001100100

#div r5, r4, r3
#  imm      func   opc    rt     rb     ra
#0b0000000 00011 00000 00101 00011 00100
0b00000000001100000001010001100100

#divu r5, r4, r3
#  imm      func   opc    rt     rb     ra
#0b0000000 00100 00000 00101 00011 00100
0b00000000010000000001010001100100

#bra r5
#  imm      func   opc    rt     rb     ra
#0b0000000 00000 00100 00101 00000 00000
0b00000000000000100001010000000000

# this is the end
```

Hex



```
# this is a comment
#add r5, r4, r3
#    rt, ra, rb
#  imm      func   opc    rt     rb     ra
#0b0000000 00000 00000 00101 00011 00100
#0b00000000000000000001010001100100
0x00001464

#add r3, r2, r1
#  imm      func   opc    rt     rb     ra
#0b0000000 00000 00000 00011 00001 00010
#0b00000000000000000000110000100010
0x00000C22

#mul r5, r4, r3
#  imm      func   opc    rt     rb     ra
#0b0000000 00010 00000 00101 00011 00100
#0b00000000001000000001010001100100
0x00201464

#sub r5, r4, r3
#  imm      func   opc    rt     rb     ra
#0b0000000 00001 00000 00101 00011 00100
#0b00000000000100000001010001100100
0x00101464

#div r5, r4, r3
#  imm      func   opc    rt     rb     ra
#0b0000000 00011 00000 00101 00011 00100
#0b00000000001100000001010001100100
0x00301464

#divu r5, r4, r3
#  imm      func   opc    rt     rb     ra
#0b0000000 00100 00000 00101 00011 00100
#0b00000000010000000001010001100100
0x00401464
```

DHDT Info Options

- --help : Prints the help menu

```
$> dhdt --help
```

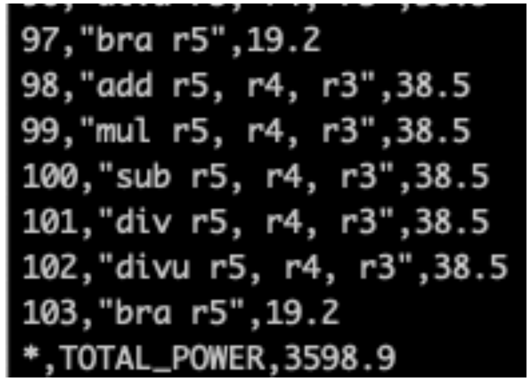
DHDT Execution Options

- --coregen /path/to/coregen.yaml: Utilize the target CoreGen IR
- --llvm /path/to/stonecutter.ir: Utilize the target StoneCutter LLVM IR
- --inst /path/to/trace.inst: Instruction trace input
- Execution types:
  - --hazard : Analyze hazards between instructions
  - --power : Analyze the power across individual instructions

```
$> dhdt --coregen ./BasicRISC.yaml --llvm ./BasicRISC.sc.ir --inst test.inst --hazard
$> dhdt --coregen ./BasicRISC.yaml --llvm ./BasicRISC.sc.ir --inst test.inst --power
```

DHDT Graph Options

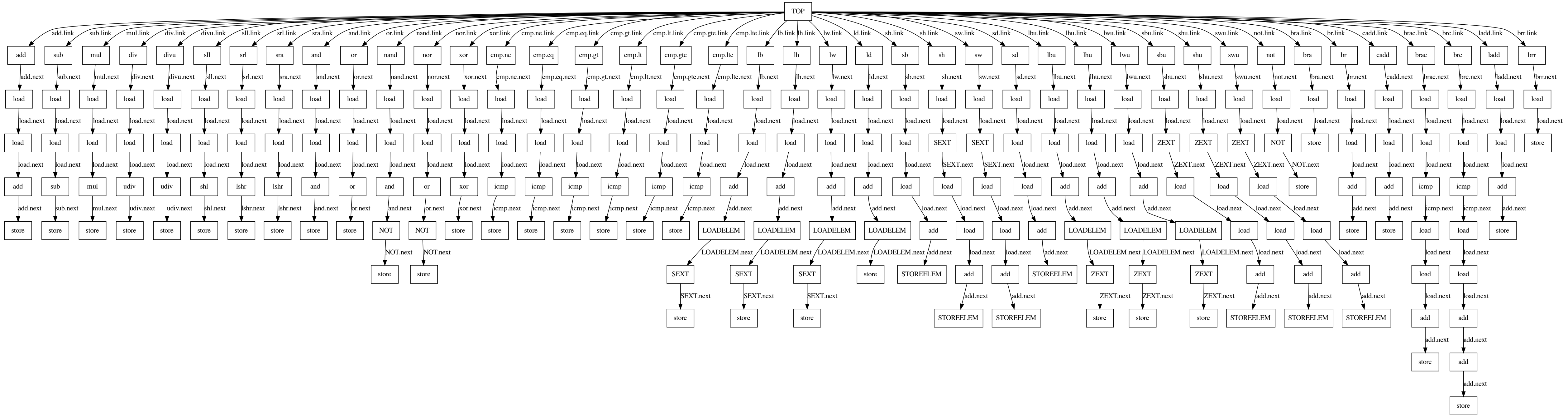
- --dot /path/to/graph.dot : Outputs the internal graph structure to a GraphViz DOT file
  - This is very handy to investigate idiosyncrasies in the power/hazard output
- --output /path/to/output.csv: Outputs the final data to a CSV file



```
97,"bra r5",19.2
98,"add r5, r4, r3",38.5
99,"mul r5, r4, r3",38.5
100,"sub r5, r4, r3",38.5
101,"div r5, r4, r3",38.5
102,"divu r5, r4, r3",38.5
103,"bra r5",19.2
*,TOTAL_POWER,3598.9
```

```
$> dhdt --dot test.dot --coregen ./BasicRISC.yaml --llvm ./BasicRISC.sc.ir --power -
-inst trace.inst
$> dot -Tpdf test.dot > test.pdf
$> dhdt --coregen ./BasicRISC.yaml --llvm ./BasicRISC.sc.ir --power --inst trace.inst
--output out.csv
```

DHDT Internal ISA Graph


TOP
add.link add add.next load load.next load load.next add add.next store
sub.link sub sub.next load load.next load load.next sub sub.next store
mul.link mul mul.next load load.next load load.next mul mul.next store
div.link div div.next load load.next load load.next udiv udiv.next store
divu.link divu divu.next load load.next load load.next udiv udiv.next store
sll.link sll sll.next load load.next load load.next shl shl.next store
srl.link srl srl.next load load.next load load.next lshr lshr.next store
sra.link sra sra.next load load.next load load.next lshr lshr.next store
and.link and and.next load load.next load load.next and and.next store
or.link or or.next load load.next load load.next or or.next store
nand.link nand nand.next load load.next load load.next and and.next NOT NOT.next store
nor.link nor nor.next load load.next load load.next or or.next NOT NOT.next store
xor.link xor xor.next load load.next load load.next xor xor.next store
cmp.ne.link cmp.ne cmp.ne.next load load.next load load.next icmp icmp.next store
cmp.eq.link cmp.eq cmp.eq.next load load.next load load.next icmp icmp.next store
cmp.gt.link cmp.gt cmp.gt.next load load.next load load.next icmp icmp.next store
cmp.lt.link cmp.lt cmp.lt.next load load.next load load.next icmp icmp.next store
cmp.gte.link cmp.gte cmp.gte.next load load.next load load.next icmp icmp.next store
cmp.lte.link cmp.lte cmp.lte.next load load.next load load.next icmp icmp.next store
lb.link lb lb.next load load.next load load.next add add.next LOADELEM LOADELEM.next SEXT SEXT.next store
lh.link lh lh.next load load.next load load.next add add.next LOADELEM LOADELEM.next SEXT SEXT.next store
lw.link lw lw.next load load.next load load.next add add.next LOADELEM LOADELEM.next SEXT SEXT.next store
ld.link ld ld.next load load.next load load.next add add.next LOADELEM LOADELEM.next store
sb.link sb sb.next load load.next load load.next load load.next add add.next STOREELEM
sh.link sh sh.next load load.next SEXT SEXT.next load load.next add add.next STOREELEM
sw.link sw sw.next load load.next SEXT SEXT.next load load.next add add.next STOREELEM
sd.link sd sd.next load load.next load load.next load load.next add add.next STOREELEM
lbu.link lbu lbu.next load load.next load load.next add add.next LOADELEM LOADELEM.next ZEXT ZEXT.next store
lhu.link lhu lhu.next load load.next load load.next add add.next LOADELEM LOADELEM.next ZEXT ZEXT.next store
lwu.link lwu lwu.next load load.next load load.next add add.next LOADELEM LOADELEM.next ZEXT ZEXT.next store
sbu.link sbu sbu.next load load.next ZEXT ZEXT.next load load.next add add.next STOREELEM
shu.link shu shu.next load load.next ZEXT ZEXT.next load load.next add add.next STOREELEM
swu.link swu swu.next load load.next ZEXT ZEXT.next load load.next add add.next STOREELEM
not.link not not.next load load.next NOT NOT.next store
bra.link bra bra.next load load.next store
br.link br br.next load load.next load load.next add add.next store
cadd.link cadd cadd.next load load.next load load.next add add.next store
brac.link brac brac.next load load.next load load.next icmp icmp.next load load.next load load.next add add.next store
brc.link brc brc.next load load.next load load.next icmp icmp.next load load.next load load.next add add.next add add.next store
ladd.link ladd ladd.next load load.next load load.next add add.next store
brr.link brr brr.next load load.next store

Going from YAML to Power/Hazard Output

- Step 1: Use CGCLI to generate a singular StoneCutter source file
- Step 2: Compile the StoneCutter file to (optimized) LLVM IR
- Step 3: Use DHDT to analyze the power/hazard logic

We highly suggest using a Makefile to automate this process!

```
$> cgcli --root ./BasicRISC --ir ./BasicRISC/BasicRISC.yaml --stonecutter
$> sccomp -D --keep -O3 ./BasicRISC/RTL/stonecutter/stonecutter.sc
$> dhdt --coregen ./BasicRISC/BasicRISC.yaml --llvm
./BasicRISC/RTL/stonecutter/stonecutter.sc.ll --inst trace.inst --power
```

What can you do with DHDT?

- Example 1: Evaluate a design based upon known power values (using a power template)
- Example 2: Evaluate a design against two different power templates that represent different fabs/reticles
- Example 3: Evaluate the power of different instruction traces
- Example 4: Evaluate two different hardware implementations of the same ISA against the same power template (design trade-off)
- Example 5: Evaluate the hazards between multiple instructions

Things to remember…

- The DHDT tool has a simple crack/decode interface
  - When using assembly, DHDT assumes that the CoreGen YAML file has `Syntax` entries for each instruction
  - The Hex and Binary decode interfaces do not require Syntax interfaces (useful for VLIW stages)
- The "simulation" of each instruction payload is rudimentary
  - We traverse the subgraph for each instruction payload and accumulate the power for each instruction
  - We do not, however, compute bit-level changes of state while we traverse the graph
  - The simulation is NOT bit-level accurate (only an approximation)

References

Where do I find more info?

Web Links

- System Architect Public Web
  - http://www.systemarchitect.tech/
- Documentation
  - Latest StoneCutter Specification:
    - http://www.systemarchitect.tech/index.php/stonecutter-language-spec/
- Tutorials
  - http://www.systemarchitect.tech/index.php/tutorials/
  - https://github.com/opensocsysarch/CoreGenTutorials

Source Code

- Main source code hosted on Github:
  - https://github.com/opensocsysarch

- CoreGen Infrastructure
  - https://github.com/opensocsysarch/CoreGen
- CoreGenPortal GUI
  - https://github.com/opensocsysarch/CoreGenPortal
- CoreGen IR Spec
  - https://github.com/opensocsysarch/CoreGenIRSpec
- StoneCutter Language Spec
  - https://github.com/opensysarch/StoneCutterLanguageSpec
- System Architect Weekly Development Releases
  - https://github.com/opensocsysarch/SystemArchitectRelease

Contact

- Issues should be submitted through the respective Github issues pages (see source code links)

- Mailing Lists:
  - http://www.systemarchitect.tech/index.php/lists/

- Direct developer contacts
  - John Leidel: jleidel<at>tactcomplabs<dot>com
  - David Donofrio: ddonofrio<at>tactcomplabs<dot>com
  - Ryan Kabrick: rkabrick<at>tactcomplabs<dot>com