C-Class Core: A walkthrough

SHAKTI Group | CSE Dept | RISE Lab | IIT Madras

Presenter: Neel Gala

What is C-Class ?

- C-Class is an RTL implementation of the RISC-V spec!
- Is it a chip?
  - Nope! It is available as a design written in Bluespec Spec System Verilog
- Where can I buy it?
  - You don't have to - Its completely open-source under BSD license!
- Where can I download it?
  - https://gitlab.com/shaktiproject/cores/c-class
- Can I build a chip from it?
  - Absolutely
- Has it been taped-out?
  - Yes - Twice !! - both as test-chips
  - Intel 22nm and SCL 180nm
- Can I evaluate it on FPGA?
  - Most definitely - https://gitlab.com/shaktiproject/cores/shakti-soc
- Can I simulate it?
  - Yes - using open-source verilator or commercial simulators as well
- Are there commercial chips available of the C-class?
  - Not yet - Hope to have some soon!
  - Many private and strategic bodies are evaluating/deploying C-class as of today.

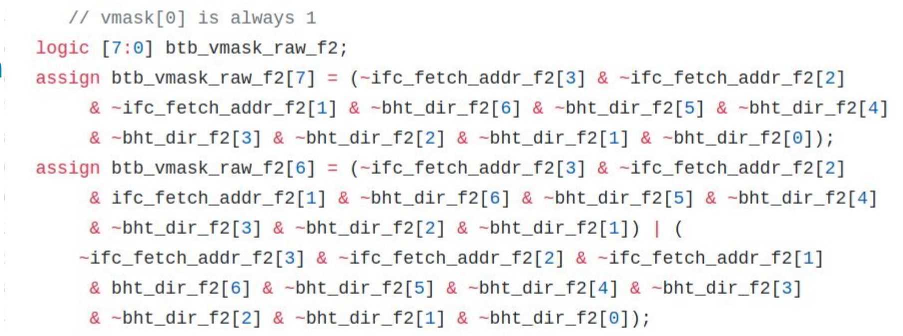
Why Bluespec?

- Strong library of commonly used blocks
- Automates significant amount of glue-logic for resource-resolution
- Guaranteed Synthesis
- High degree of parametrization - Strongly type checked
- Productivity is more than doubled !

The Generated RTL is not human readable?

Bluespec Generated verilog is much more structured than what a hum
would write!

- Register, wire names are retained to a maximum extent.
- All registers have a D_IN and EN signal. Easy to track.
- All rules have a corresponding Will_fire and Can_fire signal.
- Can be used for UVM based methodology!

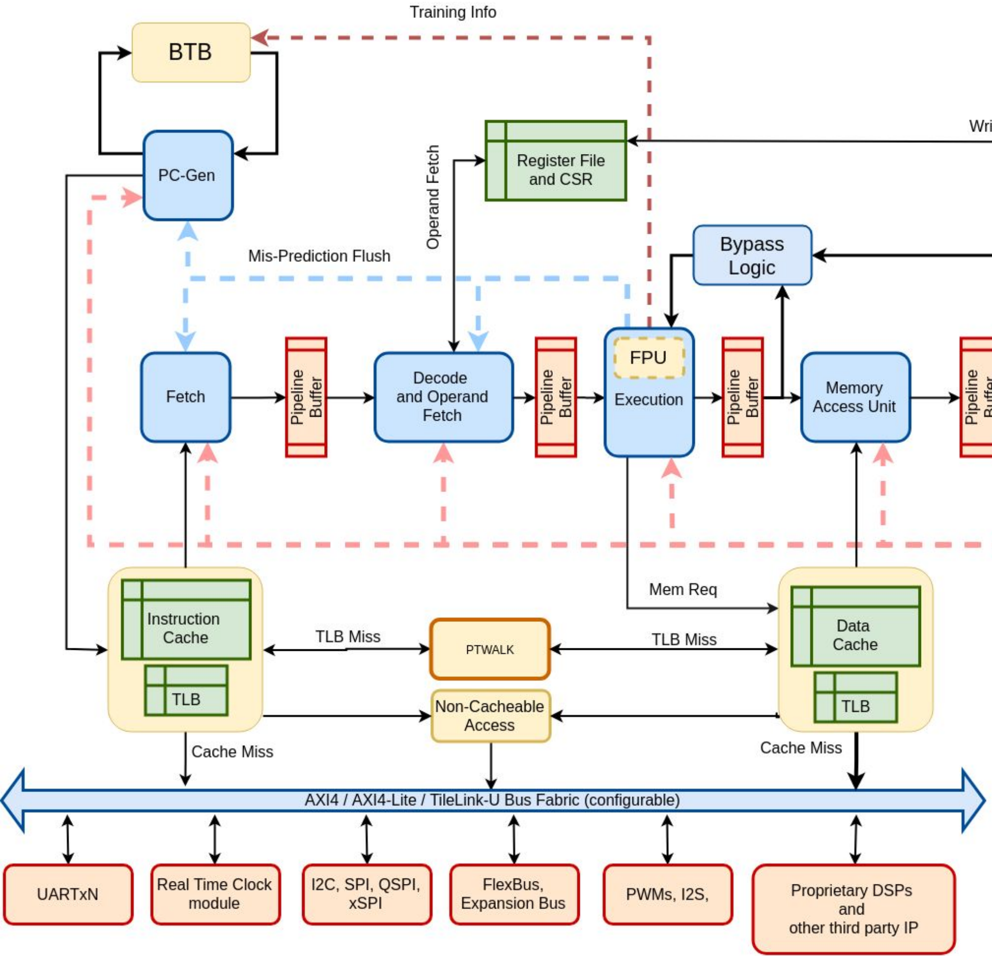


```
   // vmask[0] is always 1
logic [7:0] btb_vmask_raw_f2;
assign btb_vmask_raw_f2[7] = (~ifc_fetch_addr_f2[3] & ~ifc_fetch_addr_f2[2]
     & ~ifc_fetch_addr_f2[1] & ~bht_dir_f2[6] & ~bht_dir_f2[5] & ~bht_dir_f2[4]
     & ~bht_dir_f2[3] & ~bht_dir_f2[2] & ~bht_dir_f2[1] & ~bht_dir_f2[0]);
assign btb_vmask_raw_f2[6] = (~ifc_fetch_addr_f2[3] & ~ifc_fetch_addr_f2[2]
     & ifc_fetch_addr_f2[1] & ~bht_dir_f2[6] & ~bht_dir_f2[5] & ~bht_dir_f2[4]
     & ~bht_dir_f2[3] & ~bht_dir_f2[2] & ~bht_dir_f2[1]) | (
    ~ifc_fetch_addr_f2[3] & ~ifc_fetch_addr_f2[2] & ~ifc_fetch_addr_f2[1]
     & bht_dir_f2[6] & ~bht_dir_f2[5] & ~bht_dir_f2[4] & ~bht_dir_f2[3]
     & ~bht_dir_f2[2] & ~bht_dir_f2[1] & ~bht_dir_f2[0]);
```

Feature of C-class

- Amongst one of the most configurable RISC-V Cores.
- A Simple 5 stage in-order 32/64 bit core
- ISA Support: **RV[32/64]I[MAFDCSUN]**
- Supervisor Support: sv32, sv38 and sv48
- Parameterized and optimized Single/Double Precision Floating point support.
- Supports RISC-V Debug Spec - with triggers
- Includes an optional Branch predictor and Return-Address-Stack
- Parameterized I and D $ with optional ECC support
- Can be configured for different fabrics : AXI4, AXI4-Lite, TileLink, etc.
- Can boot Linux and RTOS

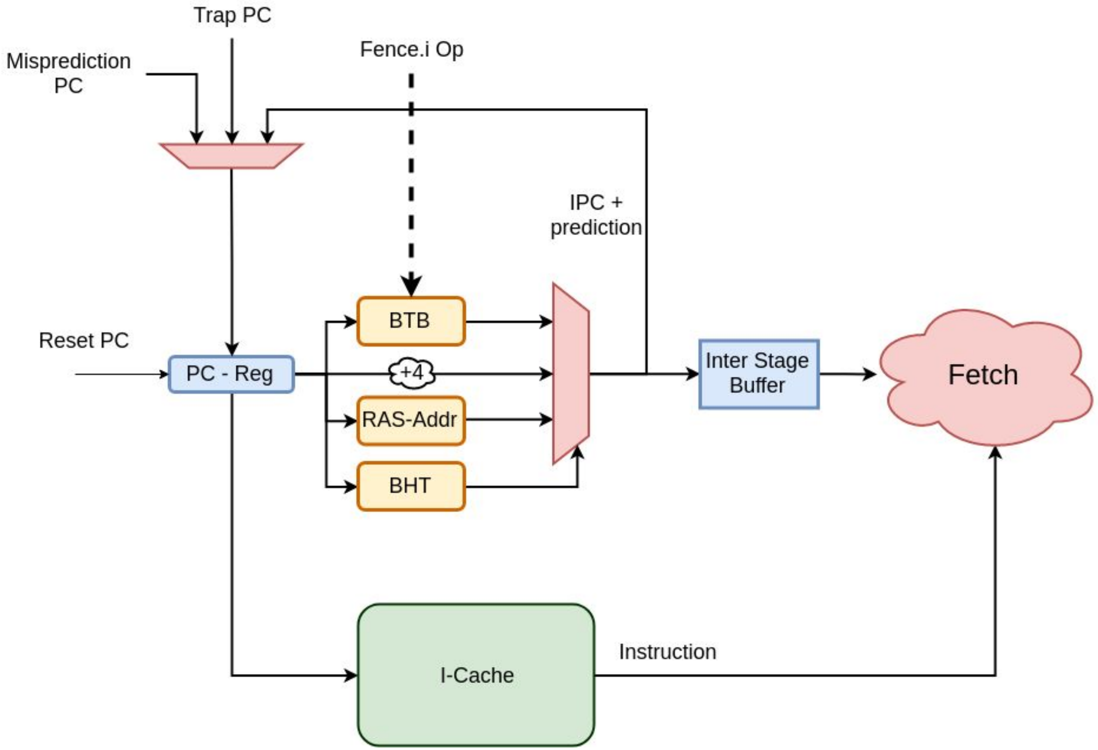
Micro Architecture



Optional Modules:

- Branch Predictor
- Return Address Stack
- Instruction Cache
- Data Cache
- Floating Point Unit
- PTWalk (only when Supervisor enabled)

PC - Gen Stage

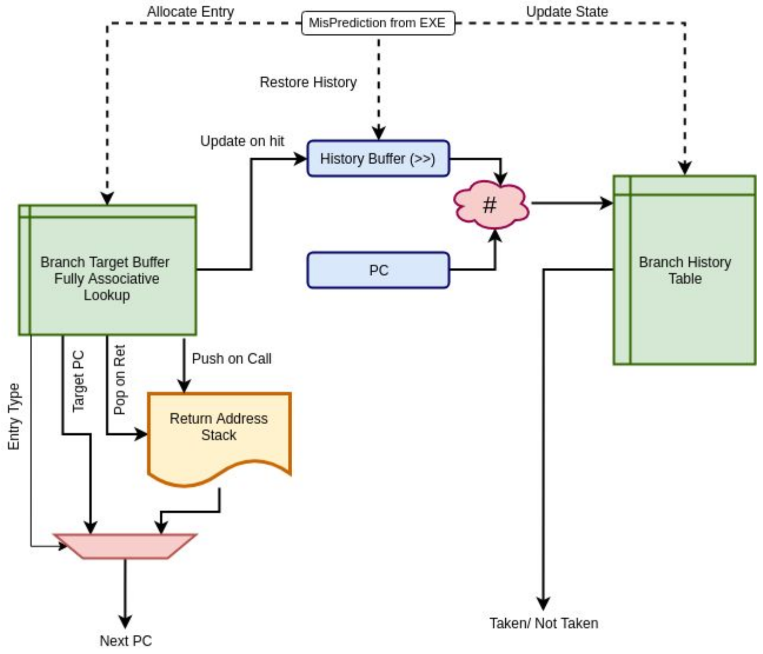


Compressed support makes things tricky

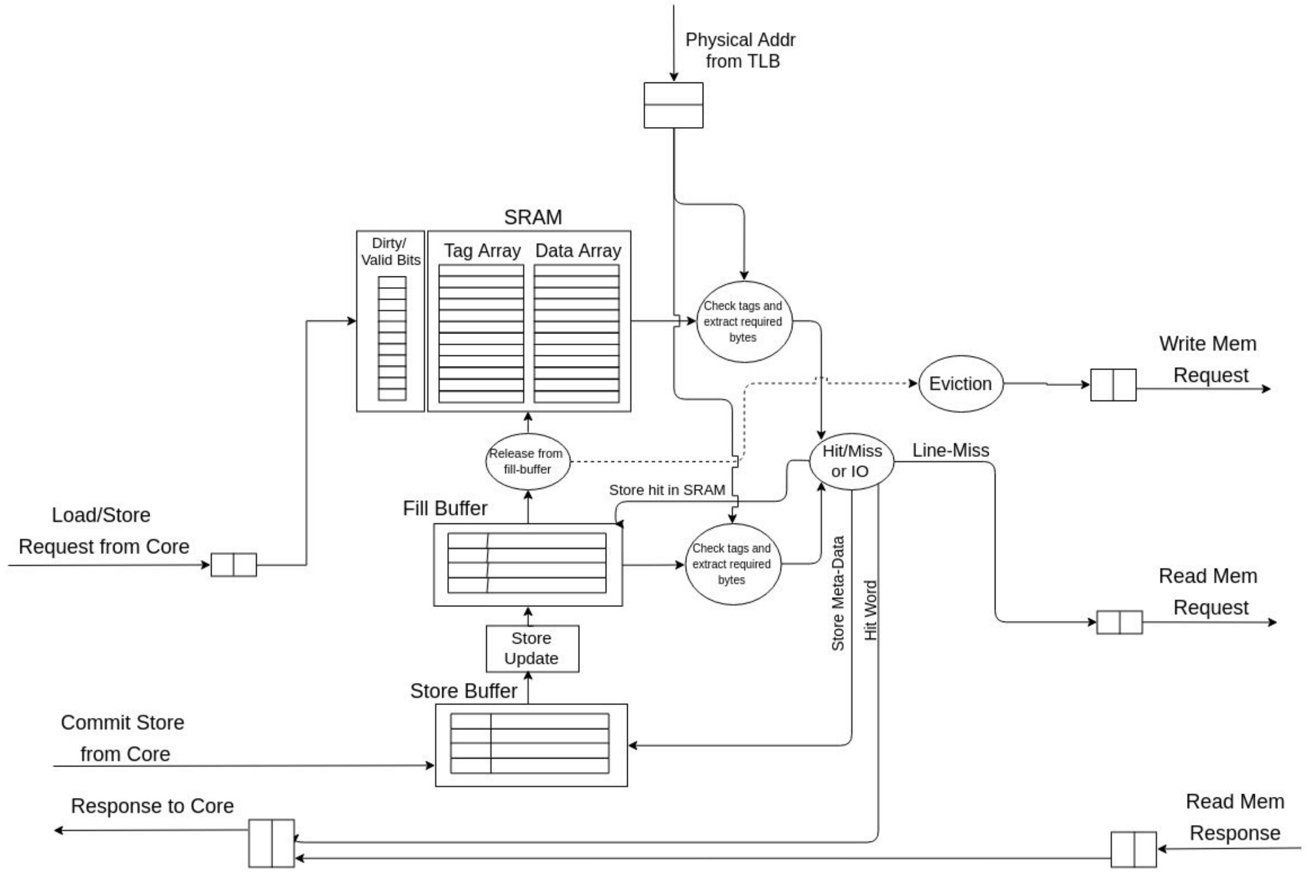
- The cache and bpu always receive 4-byte aligned addresses - PC4
- The BPU will respond with 2 predictions - for PC and PC+2.
- If PC+2 is a 4-byte instruction (i.e. not compressed) and is predicted taken to jump to NPC we first need to send PC+4 to receive the upper 16 bits from the cache and then jump to NPC

PC Gen Stage: Branch Predictor

- C-class currently has a GSHARE branch predictor
- A fully-associative branch target buffer
- Leverages the optimized One-Hot Indexing mechanism for vectors in BSV.
- An indexed 2-bit branch history table
- Maintain 3 bits for speculative history
- BTB holds type of Entry:
  - Ret - pop from RAS
  - Call - push pc+2/4 to RAS
  - Branch
- BTB also holds a bit to indicate edge case.



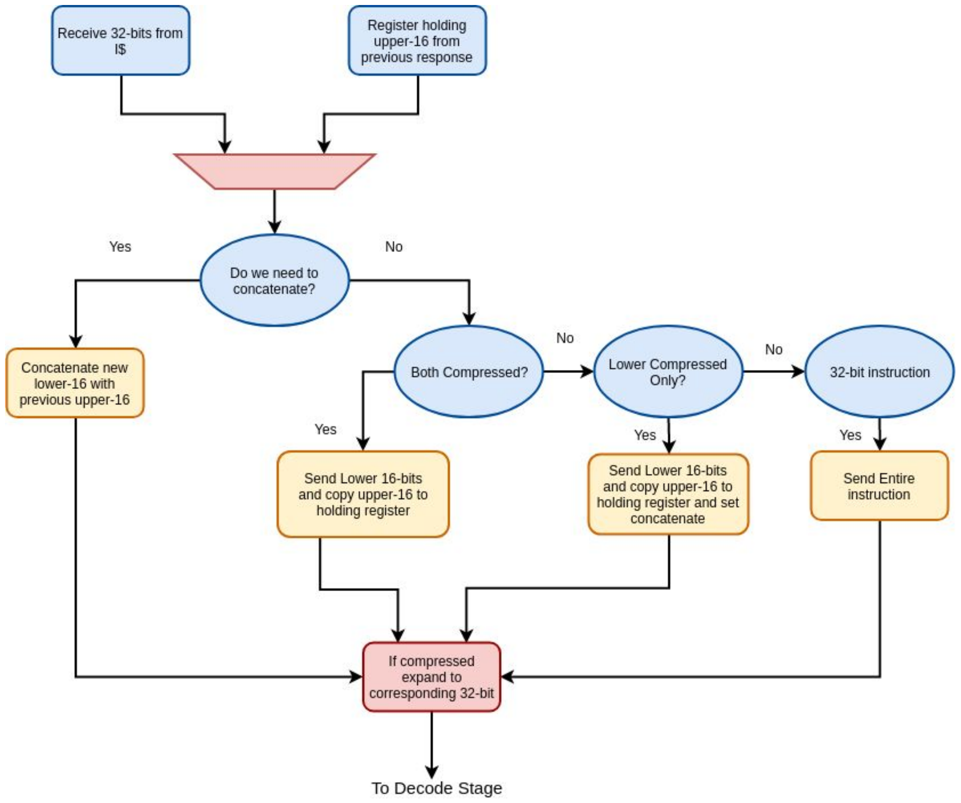
L1-Cache


Physical Addr from TLB
SRAM
Dirty/ Valid Bits
Tag Array
Data Array
Check tags and extract required bytes
Write Mem Request
Eviction
Hit/Miss or IO
Line-Miss
Release from fill-buffer
Store hit in SRAM
Fill Buffer
Load/Store Request from Core
Check tags and extract required bytes
Store Meta-Data
Hit Word
Read Mem Request
Store Update
Store Buffer
Commit Store from Core
Response to Core
Read Mem Response

Fetch 32-bits at a time from the Cache

- **Case-1**: entire word is a 32-bit instruction. In this case the entire word and the prediction for pc is sent to the decode stage.
- **Case-2**: word contains 2 16-bit instructions. in this case in the first cycle the lower 16-bits of the word and prediction of pc is sent to the decode stage. In the next cycle the upper 16-bits and prediction of pc+2 is sent to the decode stage.
- **Case-3:** lower 16-bits need to be concatenated with the upper 16-bits of the previous I$ response. in this case the a new 32-bit instruction is formed and the prediction of the previous response is sent to the decode stage.
- **Case-4:** Only the upper 16-bits of the I$ needs to be analysed. If the upper 16-bits are compressed then the same and prediction of pc+2 is sent to the decode stage. If however, the upper 16-bits are the lower part of a 32-bit instruction, then we need to wait for the next I$ response and use the Case-3 scheme then. Now one can land in this case, when there is jump to a 32-bit instruction placed at a 2-byte boundary.

Fetch stage with compressed support !



Receive 32-bits from I$
Register holding upper-16 from previous response
Do we need to concatenate?
Yes
No
Concatenate new lower-16 with previous upper-16
Both Compressed?
No
Lower Compressed Only?
No
32-bit instruction
Yes
Send Lower 16-bits and copy upper-16 to holding register
Yes
Send Lower 16-bits and copy upper-16 to holding register and set concatenate
Yes
Send Entire instruction
If compressed expand to corresponding 32-bit
To Decode Stage

Consider the following snippet:

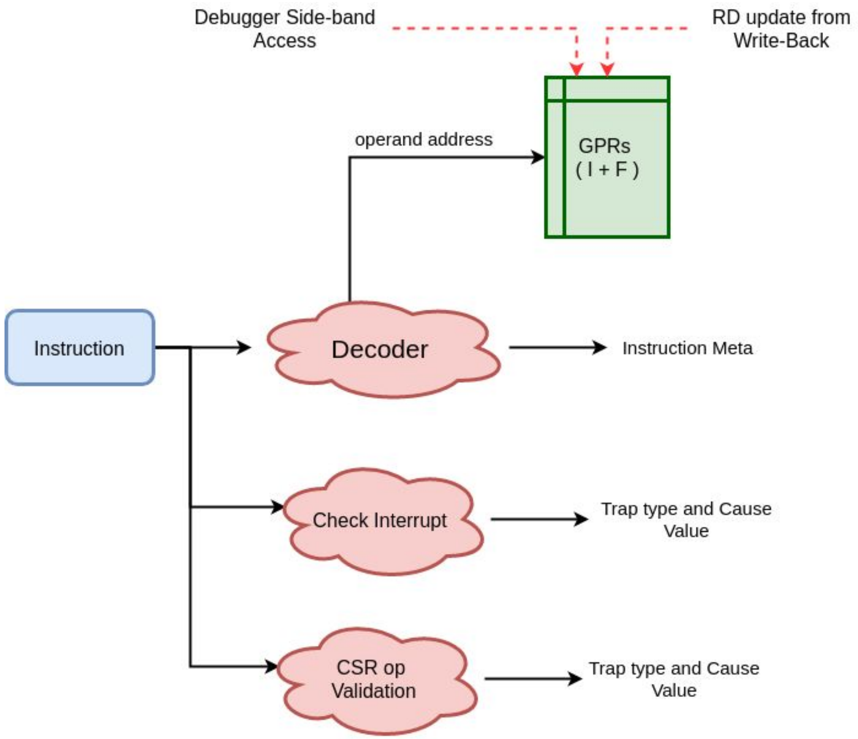
```
8000106e: 0x00001797          auipc a5,0x1
...
800010d8: 0xF97FF0eF          jal ra,8000106e
```

Even if all the code resides in the I$ this scenario would still lead to a single cycle delay.

Now imagine this scenario occurring thrice within a single iteration of Dhrystone?

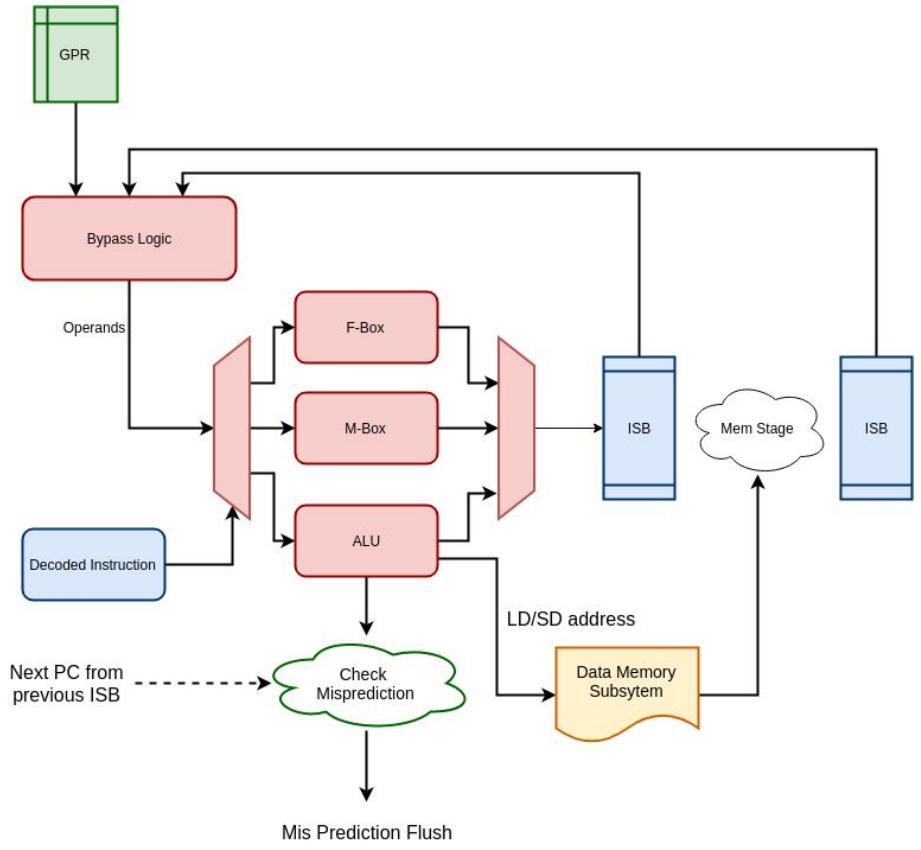
Compressed only guarantees code-density not performance !

Decode Stage


Debugger Side-band Access
RD update from Write-Back
operand address
GPRs (I + F)
Instruction
Decoder
Instruction Meta
Check Interrupt
Trap type and Cause Value
CSR op Validation
Trap type and Cause Value

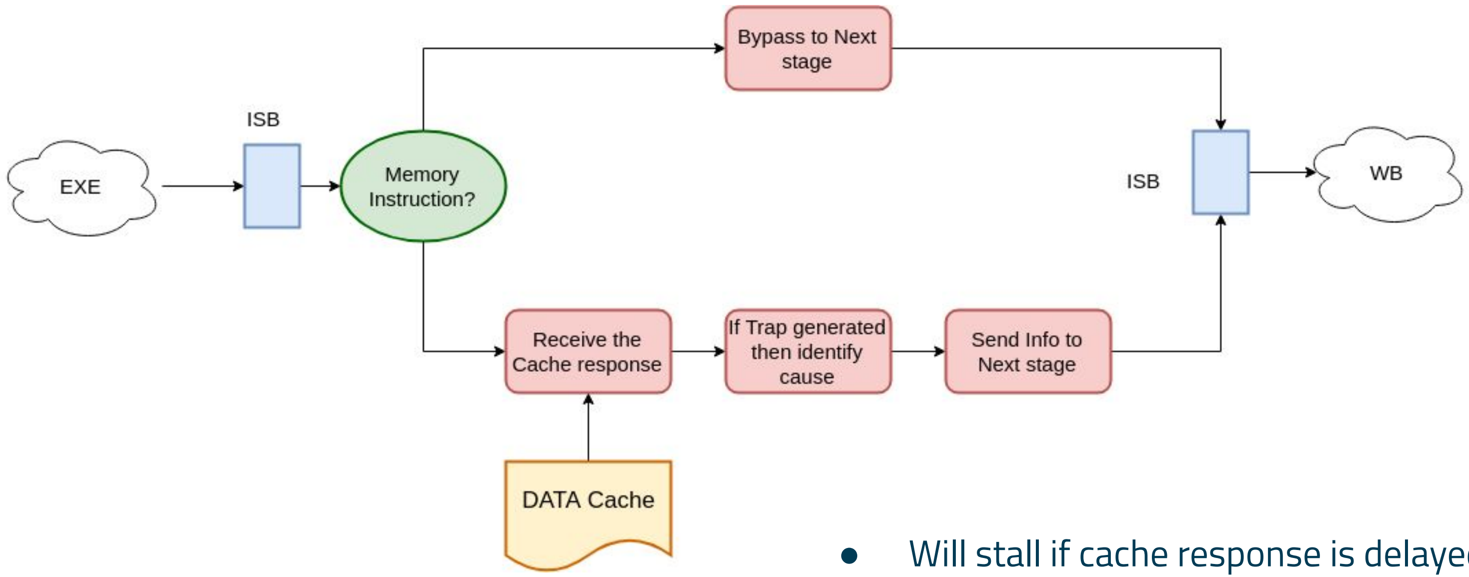
- Operand fetch happens in this stage
- RF fwds the latest commit being performed in the same cycle
- Interrupts are also checked here
- CSR ops are checked for access-violations
- Fence.i and Sfence tag the next instruction to be re-run : we allocate unused exception cause values for this feature.
- Debug request to halt, step or resume are also captured here as interrupts - discussed later
- WFI will stall the pipe here and wait for an interrupt

Execute Stage



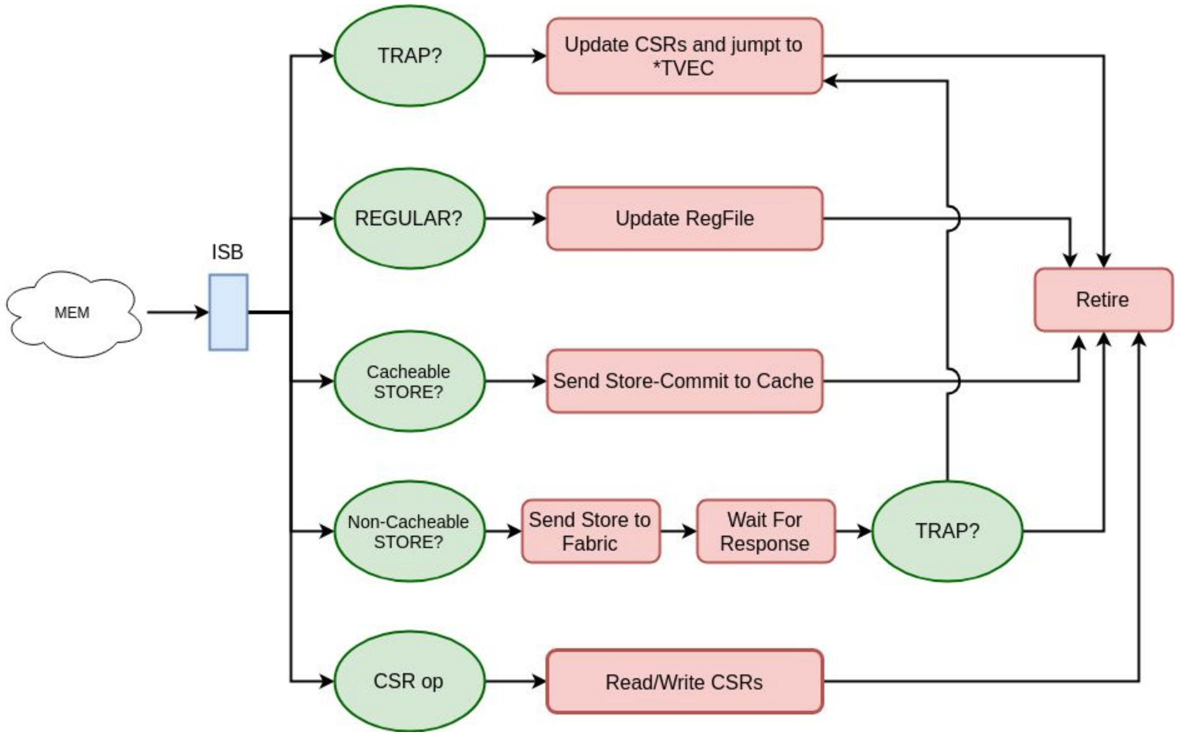
- In order execution
- F-Box and M-Box are multicycle ops
- M-Box:
  - Parameterized stages and depends on re-timing for best closure
  - Iterative non-restoring division algorithm
- F-Box:
  - Currently uses hand-optimized iterative algorithms
  - Working on retimed based modules for better frequency
- ALU generates mis-aligned traps as well
- Mis prediction is checked in this stage.
- Bypass logic is small and simple

Memory Stage



- Will stall if cache response is delayed
- Captures trap from caches

Write-Back Stage

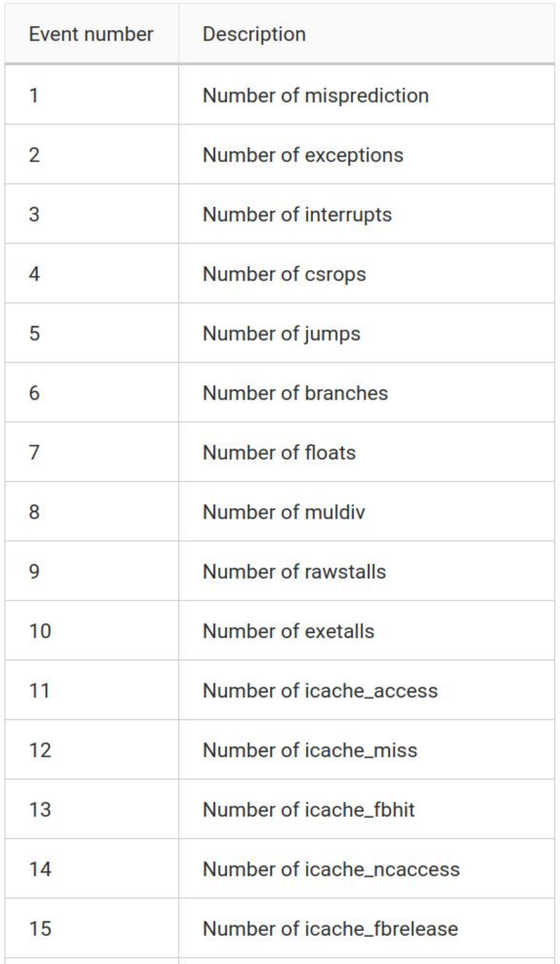
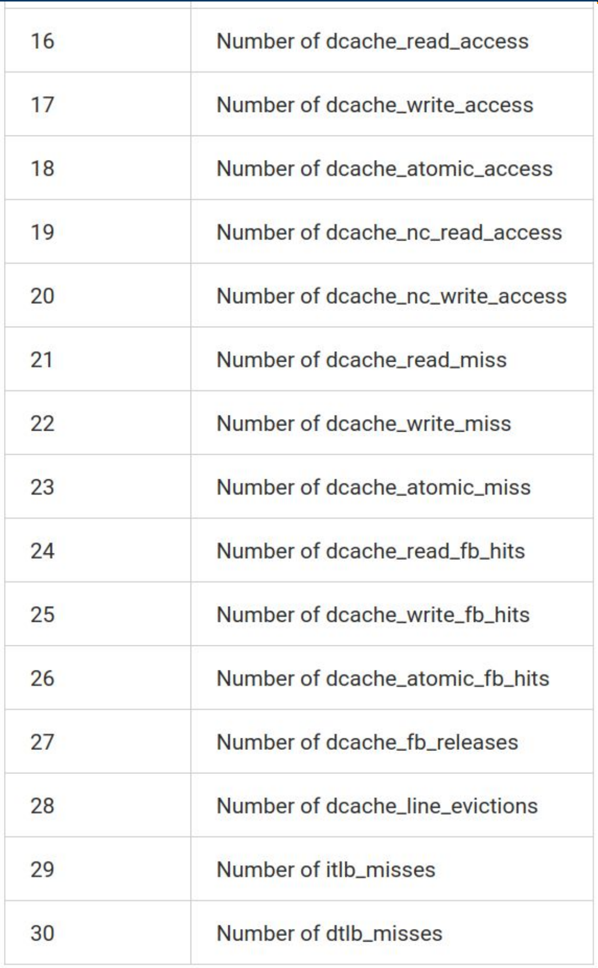


- Cacheable Stores do not stall
- Non Cacheable Stores will stall and trap if necessary

Debugger Support

1. C-Class has a debugger based on the riscv-debug-spec -v0.14
2. Trigger support is present as well
    a. Number of triggers - parameterized
3. A simple halt loop is used to indicate halted state
4. Resume and Halt requests from the Debugger are received via custom interrupts to the core.
    a. Re-use already existing circuitry to jump to halt-loop
5. A BSV based JTAG Tap is also available - silicon verified
6. Xilinx BSCANE based JTAG tap can also be used to interface with the Debugger
    a. Modified openocd to support this feature
    b. Avoids dependency on external JTAG cables and debuggers for FPGA prototyping on Xilinx.

Performance Counters/Events

Event number	Description
1	Number of misprediction
2	Number of exceptions
3	Number of interrupts
4	Number of csrops
5	Number of jumps
6	Number of branches
7	Number of floats
8	Number of muldiv
9	Number of rawstalls
10	Number of exetalls
11	Number of icache_access
12	Number of icache_miss
13	Number of icache_fbhit
14	Number of icache_ncaccess
15	Number of icache_fbrelease
16	Number of dcache_read_access
17	Number of dcache_write_access
18	Number of dcache_atomic_access
19	Number of dcache_nc_read_access
20	Number of dcache_nc_write_access
21	Number of dcache_read_miss
22	Number of dcache_write_miss
23	Number of dcache_atomic_miss
24	Number of dcache_read_fb_hits
25	Number of dcache_write_fb_hits
26	Number of dcache_atomic_fb_hits
27	Number of dcache_fb_releases
28	Number of dcache_line_evictions
29	Number of itlb_misses
30	Number of dtlb_misses

Interrupts for Counters:

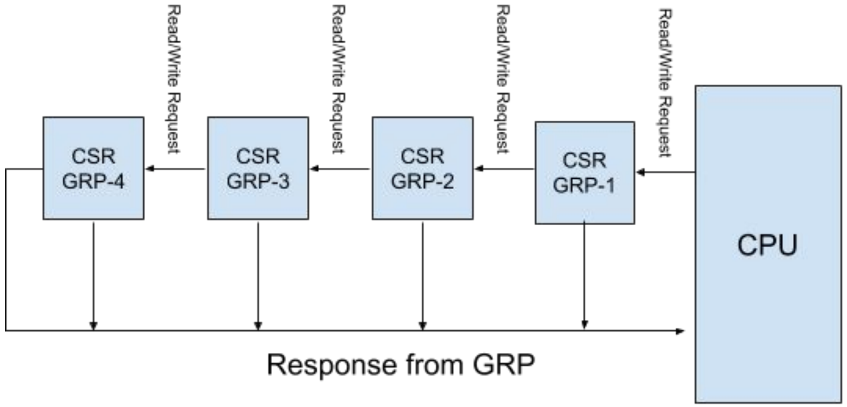
mhpm-interrupt-en:

- The encoding for this csr is the same as that of mcounteren/mcountinhibit.
- This is a read-write CSR.
- When a particular bit is set, it indicates that the corresponding counter will generate an interrupt when **the value reaches 0 and the counter is enabled**
- The interrupt can be disabled by writing a 0 to the corresponding **mhpmevent** register (equivalent to disabling the counter)

CSR Daisy Chain Grouping

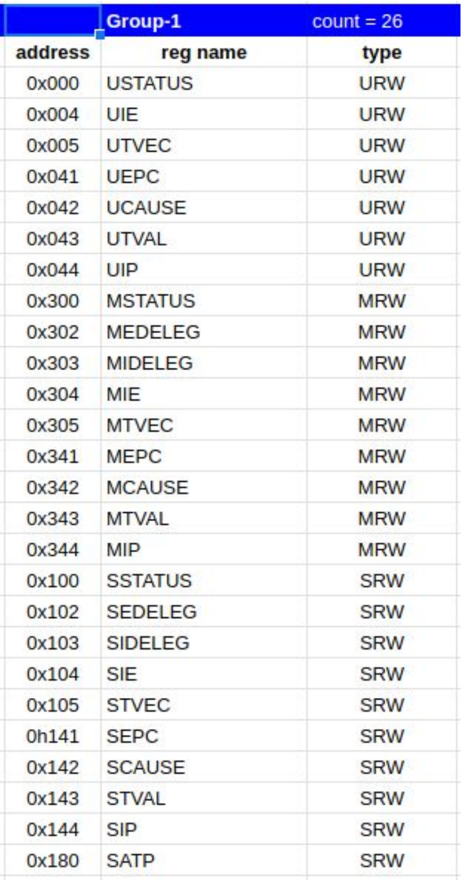
Daisy Chain of CSRs

- We propose to have 7 groups to keep a balance of csr access in each group!
- Latency:
  - Min: 1 cycles
  - Max: 7 cycles
- Not all 7 groups required
  - Parameterized to include only what is required and defined at compile time.

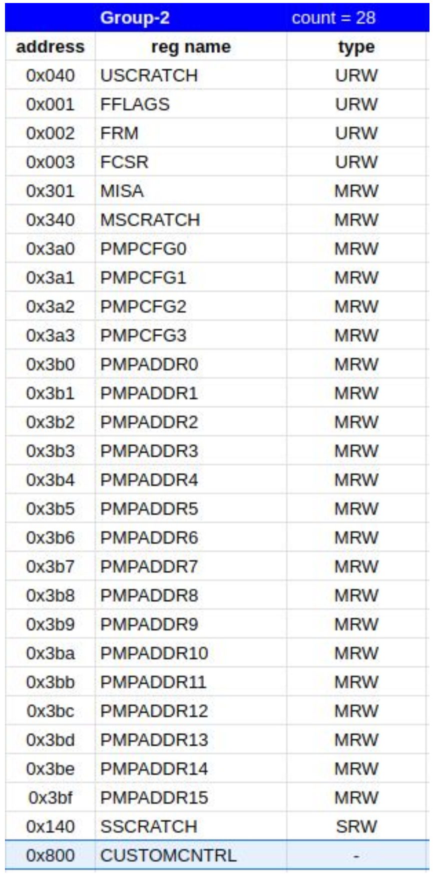


CSR Daisy Chain Grouping

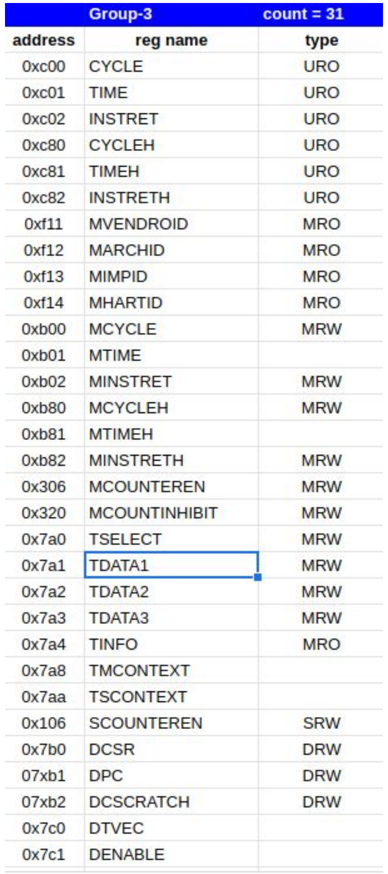
Grouping of CSRs



	Group-1	count = 26
address	reg name	type
0x000	USTATUS	URW
0x004	UIE	URW
0x005	UTVEC	URW
0x041	UEPC	URW
0x042	UCAUSE	URW
0x043	UTVAL	URW
0x044	UIP	URW
0x300	MSTATUS	MRW
0x302	MEDELEG	MRW
0x303	MIDELEG	MRW
0x304	MIE	MRW
0x305	MTVEC	MRW
0x341	MEPC	MRW
0x342	MCAUSE	MRW
0x343	MTVAL	MRW
0x344	MIP	MRW
0x100	SSTATUS	SRW
0x102	SEDELEG	SRW
0x103	SIDELEG	SRW
0x104	SIE	SRW
0x105	STVEC	SRW
0h141	SEPC	SRW
0x142	SCAUSE	SRW
0x143	STVAL	SRW
0x144	SIP	SRW
0x180	SATP	SRW



	Group-2	count = 28
address	reg name	type
0x040	USCRATCH	URW
0x001	FFLAGS	URW
0x002	FRM	URW
0x003	FCSR	URW
0x301	MISA	MRW
0x340	MSCRATCH	MRW
0x3a0	PMPCFG0	MRW
0x3a1	PMPCFG1	MRW
0x3a2	PMPCFG2	MRW
0x3a3	PMPCFG3	MRW
0x3b0	PMPADDR0	MRW
0x3b1	PMPADDR1	MRW
0x3b2	PMPADDR2	MRW
0x3b3	PMPADDR3	MRW
0x3b4	PMPADDR4	MRW
0x3b5	PMPADDR5	MRW
0x3b6	PMPADDR6	MRW
0x3b7	PMPADDR7	MRW
0x3b8	PMPADDR8	MRW
0x3b9	PMPADDR9	MRW
0x3ba	PMPADDR10	MRW
0x3bb	PMPADDR11	MRW
0x3bc	PMPADDR12	MRW
0x3bd	PMPADDR13	MRW
0x3be	PMPADDR14	MRW
0x3bf	PMPADDR15	MRW
0x140	SSCRATCH	SRW
0x800	CUSTOMCNTRL	-



	Group-3	count = 31
address	reg name	type
0xc00	CYCLE	URO
0xc01	TIME	URO
0xc02	INSTRET	URO
0xc80	CYCLEH	URO
0xc81	TIMEH	URO
0xc82	INSTRETH	URO
0xf11	MVENDROID	MRO
0xf12	MARCHID	MRO
0xf13	MIMPID	MRO
0xf14	MHARTID	MRO
0xb00	MCYCLE	MRW
0xb01	MTIME	
0xb02	MINSTRET	MRW
0xb80	MCYCLEH	MRW
0xb81	MTIMEH	
0xb82	MINSTRETH	MRW
0x306	MCOUNTEREN	MRW
0x320	MCOUNTINHIBIT	MRW
0x7a0	TSELECT	MRW
0x7a1	TDATA1	MRW
0x7a2	TDATA2	MRW
0x7a3	TDATA3	MRW
0x7a4	TINFO	MRO
0x7a8	TMCONTEXT	
0x7aa	TSCONTEXT	
0x106	SCOUNTEREN	SRW
0x7b0	DCSR	DRW
07xb1	DPC	DRW
07xb2	DCSCRATCH	DRW
0x7c0	DTVEC	
0x7c1	DENABLE	

- Group-4:
  - Counteren
  - Counters 3-9
  - Events 3-9
- Group-5
  - Counters 10-16
  - Events 10-16
- Group-6
  - Counters 17-23
  - Events 17-23
- Group-7
  - Counters 24-31
  - Events 24-31

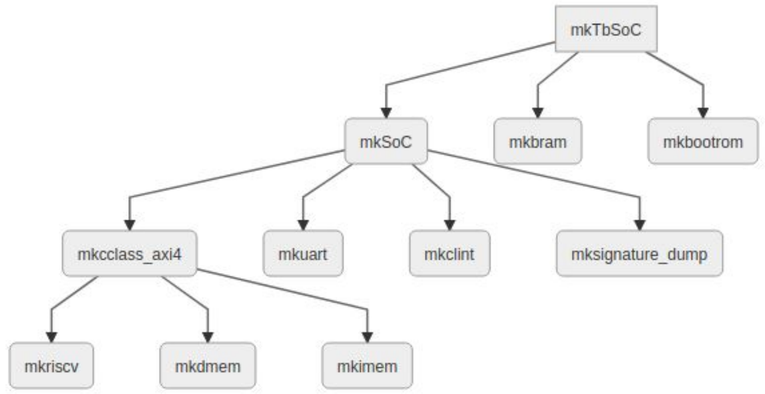
Verification

- We use cocotb for module level testing
  - Building python models for C-class units
  - Automated stress testers
  - UVM like methodology - Future Work
- Testing at the core level:
  - Every change on the master branch triggers a CI/CD which runs the riscv-tests and the compliance suite.
  - We use a series of stress-testers (a.k.a random generators)
    - AAPG
    - RISCV-TORTURE
    - CSMITH
    - RISC-V DV
  - CI/CD triggered for stress-testers triggered every night to run more than 200 tests - each test having atleast ~200K instructions minimum
    - We did find bugs through this !!

Simulating the Core!

- The repo contains a sample SoC for simulation which contains:
  - A simple UART
  - A 4KB of Boot Rom - stores the dts and initial jump vector
  - A 32MB of BRAM blocks - acts as main memory
  - A CLINT - for timer and software interrupts
  - A Signature dump module - used to write out 4-byte aligned data between two addresses into a file
- Sample TestBench is also available in BSV:
  - drive clock, reset, etc.
  - Dump UART data into a file
  - Generate instruction trace dumps

Module	Address Range
BRAM-Memory	0x80000000 - 0x8FFFFFFF
BootROM	0x00001000 - 0x00010FFF
UART	0x00011300 - 0x00011340
CLINT	0x02000000 - 0x020BFFFF
Debug-Halt Loop	0x00000000 - 0x0000000F
Signature Dump	0x00002000 - 0x0000200c



Configuring the Core

```
ISA=RV64IMAFDC

# User mode related settings
USERTRAPS=disable
USER=enable

# Supervisor related settings
SUPERVISOR=sv39
ITLBSIZE=4
DTLBSIZE=4

# Configuring M extension
MULSTAGES=2
DIVSTAGES=32

# Configuring the branch predictor
PREDICTOR=gshare
BPURESET=1
BTBDEPTH=32
BHTDEPTH=512
HISTLEN=8
EXTRAHIST=3
RASDEPTH=8

# ECC support in caches
ECC=disable

# Configuring the PMP CONFIG
PMP=disable
PMPSIZE=0
ASIDWIDTH=9

# configuring the Instruction cache
ICACHE=enable
ISETS=64
IWORDS=4
IBLOCKS=16
IWAYS=4
IFBSIZE=4
IESIZE=2
IREPL=PLRU
IRESET=1
IDBANKS=1
ITBANKS=1

# configuring the Data cache
DCACHE=enable
DESIZE=1
DSETS=64
DWORDS=8
DBLOCKS=8
DWAYS=4
DFBSIZE=8
DSBSIZE=2
DREPL=PLRU
DRESET=1
DDBANKS=1
DTBANKS=1

# Configuring Debug and Trigger Setup
TRIGGERS=0
DEBUG=disable
OPENOCD=disable

# Simulation configurations and env settings
COVERAGE=none
TRACE=disable
THREADS=1
VERILATESIM=fast
VERBOSITY=disable
RTLDUMP=enable
ASSERTIONS=enable

# miscellaneous configs
SYNTH=SIM
ARITH_TRAP=disable
RESETPC=4096
PADDR=32
COREFABRIC=AXI4

# Counter config for daisy-chain
CSRTYPE=daisy
CSR_LATENCY=low
COUNTERS_GRP4=7
COUNTERS_GRP5=7
COUNTERS_GRP6=7
COUNTERS_GRP7=8
```

A simple config file needs to be changed to configure the core based on requirements

Simulating the Core

- You need to have verilator v4.018+ installed: https://www.veripool.org/wiki/verilator
- If you have access BSC:
  - Create an instance of the core by configuring the soc_config.ini
  - Generate Verilog with Test-Bench as the top module
  - User verilator to simulate the test-bench
- If you don't have access to BSC:
  - Each commit to the master branch creates a release which contains an verilog artifact of the max configuration.
  - This can be downloaded from here: https://gitlab.com/shaktiproject/cores/c-class/blob/master/docs/verilog-artifacts/verilog-artifacts.md
- Simulation Arguments
  - Module selective logging is possible: we use a logger module which enables printing display statements which are selected as arguments during simulation
  - Instruction trace dump can also be generated if required (requires RTLDUMP=enable to be set in the config file) .

FPGA prototyping

- An instance of the C-class has been ported on Arty-A7-100t FPGA board:
  - https://gitlab.com/shaktiproject/cores/shakti-soc/tree/master/fpga/boards/artya7-100t/c-class
  - Uses Xilinx BSCANE based JTAG tap - so no external jtag cable required
- With BSV access
  - You can build your own SoC using open source devices:
    https://gitlab.com/shaktiproject/uncore/devices
  - You can modify the instance and configure the core to you requirements
- No BSV access?
  - Pre-built MCS files are also available on the repo

Benchmarking

- DMIPS : **1.72 DMIPs/ MHz**
- **ASIC Synthesis :**
  - 32-bit core can close at 400 MHz on 65nm
  - Worst case corner, 10% derating factor and 10% clock uncertainty
  - Multiplier is re-timed
- **FPGA Synthesis**
  - On Arty-A7 the 64-bit core (without caches) occupies ~6K LUTs
  - On Arty-A7 the 32-bit core (without caches) occupies ~4.5K LUTs
  - Each can clock at upto 70MHz.

Users of C-Class

- Test chip on Intel 22nm - 2017
- Test chip on SCL 180nm - 2018
- Building Safety Critical Systems - funded by Thales
- Strategic Sectors
    - Indira Gandhi Centre for Atomic Research (IGCAR)
    - Bhabha Atomic Research Centre (BARC)
    - ISRO Inertial Systems Unit (IISU)

Next in C-Class

1. POSIT Support.
    a. A Posit Enabled C-Class core
    b. Gnu toolchain support
    c. Coming Soon !!
2. Creating a Gem5 model.
    a. A cycle accurate model of the C-class
    b. Exploring System emulation support for RISC-V as well
3. Fault Tolerant Variant.
    a. Triple Lock Step cores
    b. Parameterized libraries for quickly build spatial redundancy
4. Coprocessor interface.
    a. Derived from ROCC - modified slightly
    b. FPU, Systolic Array, Bit-Manip, Crypto Cores, etc
5. Optimized FPU.
    a. Denormal support to be handled by SW instead of HW - through invisible traps
    b. Retimed and multicycle variants being designed.

Next in C-Class

1. Multi-Core variant - Cache coherency
    a. Initial draft would be Directory based MSI protocol.
    b. Using ProtoGen to build a BSV backend to generate cache and directory controllers.
2. Verification
    a. RVFI porting of C-Class
    b. More stress test generators
3. Security variants
    a. Our own TEE variants
    b. Crypto accelerators
4. Cache related updates
    a. Non blocking caches - being designed as part of I-Class
    b. 1r+1w variant
5. Hypervisor support
    a. Implementation of current draft being verified
6. Trace support
    a. Instruction trace support - Nexus based implementation
    b. Trace support based on the RISC-V WG as well

Thank you

Website: shakti.org.in

GitLab: gitlab.com/shaktiproject/cores/c-class