Macro-Op Fusion: Your RISC-V Decoder Can Do More!

India Systems' Research Discussions

Karthik B K <karthik.bk@incoresemi.com>

September 3, 2022

InCore Semiconductors

Outline

About Me

About Me

- B.Tech. from PES University, Bengaluru (pes.edu)
- RTL Design Engineer @
  InCore Semiconductors (incoresemi.com)
- Research Intern @ CHIPS (chips.pes.edu)
- I plan to do a PhD :)

Introduction

Instruction Set Architecture (ISA)

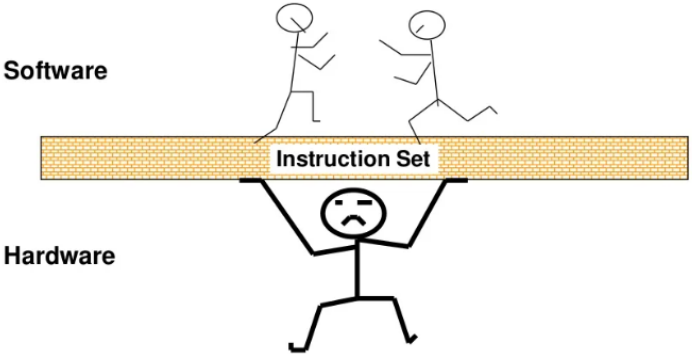
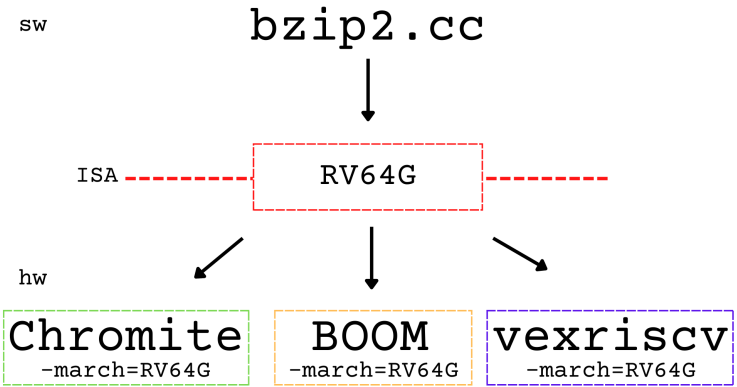


Illustration from *Hennesy & Patterson, 3rd Ed.*

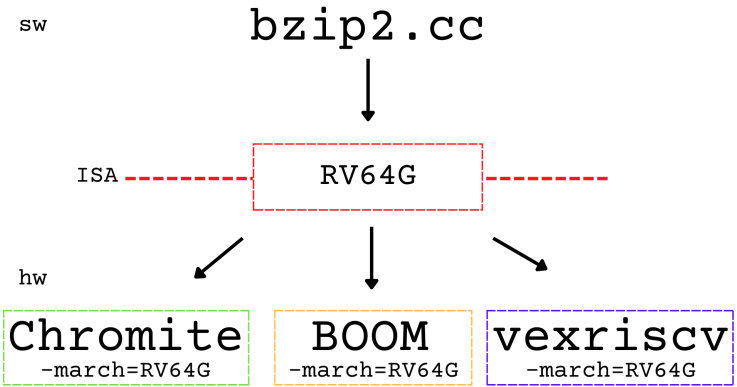
Instruction Set Architecture (ISA)

- Interface between HW and SW



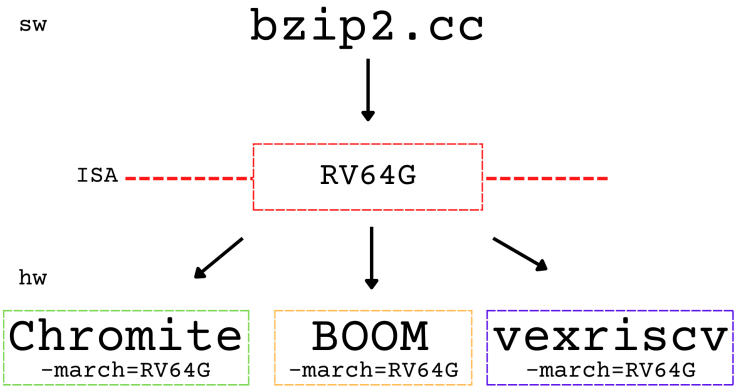
Instruction Set Architecture (ISA)

- Interface between HW and SW
- The 'language' your processor speaks



Instruction Set Architecture (ISA)

- Interface between HW and SW
- The 'language' your processor speaks
- x86, ARMv8, RISC-V



Measuring Performance

Iron Law of Performance

$$\underset{\text{(seconds/program)}}{\text{Performance}} = \frac{\text{cycles}}{\text{instr}} * \frac{\text{seconds}}{\text{cycles}} * \frac{\text{instr}}{\text{program}}$$

Iron Law of Performance

$$\underset{\text{(seconds/program)}}{\text{Performance}} = \frac{\boxed{\text{cycles}}}{\text{instr}} * \frac{\text{seconds}}{\boxed{\text{cycles}}} * \frac{\text{instr}}{\text{program}}$$

uarch

process

isa

The Journey to Better Performance (Fixed ISA)

the culprits

gcc

compiler

programmer

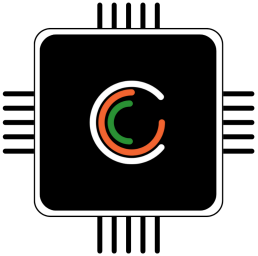
u-architect

what can they do better?

Chromite RISC-V: An Example

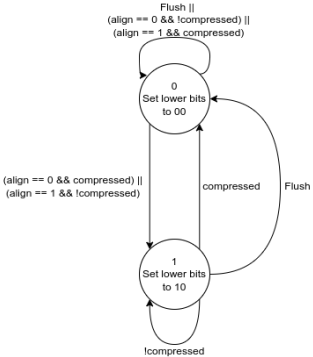
What is Chromite RISC-V?

- Open-source core generator
- 6 stage in-order pipeline
- Highly configurable



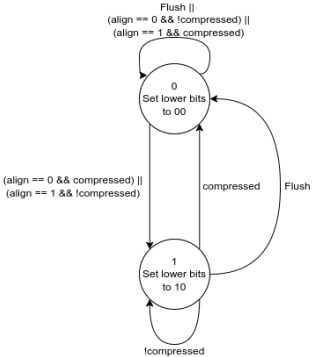
Core Pipeline: Stage 1: Fetch

- Interacts with PC Gen stage and IMS



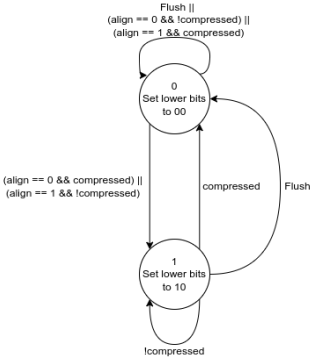
Core Pipeline: Stage 1: Fetch

- Interacts with PC Gen stage and IMS
- Decompresses all compressed instructions



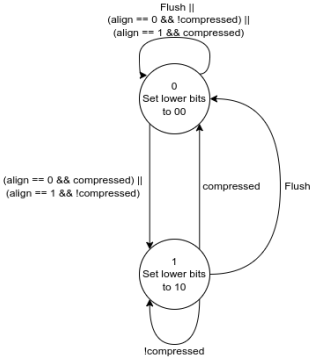
Core Pipeline: Stage 1: Fetch

- Interacts with PC Gen stage and IMS
- Decompresses all compressed instructions
- compressed: whether current instr is compressed



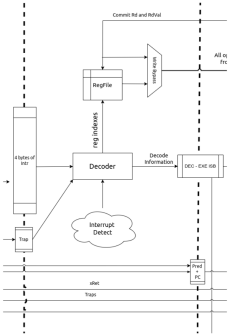
Core Pipeline: Stage 1: Fetch

- Interacts with PC Gen stage and IMS
- Decompresses all compressed instructions
- compressed: whether current instr is compressed
- align: present alignment - 2/4 byte



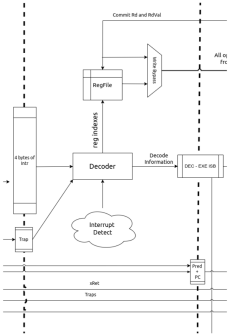
Core Pipeline: Stage 2: Decode

- Decodes the following artifacts:
  - Operand indices: rd, rs1, rs2[, rs3]



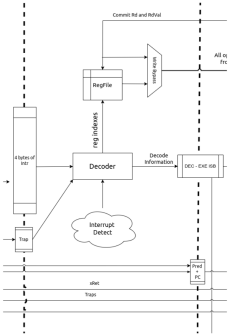
Core Pipeline: Stage 2: Decode

- Decodes the following artifacts:
  - Operand indices: rd, rs1, rs2[, rs3]
  - Immediate Value: produces a 32 bit value



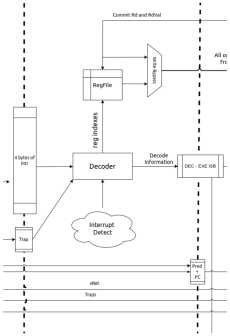
Core Pipeline: Stage 2: Decode

- Decodes the following artifacts:
  - Operand indices: rd, rs1, rs2[, rs3]
  - Immediate Value: produces a 32 bit value
  - Operand type: xrf, frf



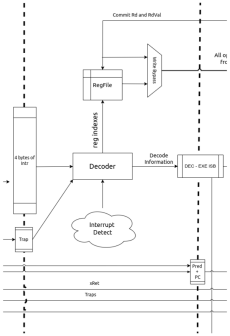
Core Pipeline: Stage 2: Decode

- Decodes the following artifacts:
  - Operand indices: rd, rs1, rs2[, rs3]
  - Immediate Value: produces a 32 bit value
  - Operand type: xrf, frf
  - Instruction Type: Arithmetic, Memory, Branch, etc



Core Pipeline: Stage 2: Decode

- Decodes the following artifacts:
  - Operand indices: rd, rs1, rs2[, rs3]
  - Immediate Value: produces a 32 bit value
  - Operand type: xrf, frf
  - Instruction Type: Arithmetic, Memory, Branch, etc
  - Function Opcode: re-encoded 7 bit field used by exec stage (ALU)



What can we do better?

u-architect

What can we do better?

- In the decoder:
  - Invisible traps - emulation routines

u-architect

What can we do better?

- In the decoder:
  - Invisible traps - emulation routines
  - Unroll instructions - micro ops

u-architect

What can we do better?

- In the decoder:
  - Invisible traps - emulation routines
  - Unroll instructions - micro ops
  - Combine instructions - macro ops

u-architect

What can we do better?

- In the decoder:
  - Invisible traps - emulation routines
  - Unroll instructions - micro ops
  - Combine instructions - macro ops
    - Superscalar only?
    - micro caches for single issue

u-architect

Optimisations to the Micro-Architecture

Micro- and Macro-Ops: What Are They?

Micro- and Macro-Ops: What Are They?

- happens in decode stage

Micro- and Macro-Ops: What Are They?

- happens in decode stage
- completely hidden from user

Micro- and Macro-Ops: What Are They?

- happens in decode stage
- completely hidden from user
- can be split and combined in different stages

Micro- and Macro-Ops: What Are They?

- happens in decode stage
- completely hidden from user
- can be split and combined in different stages
- both increase performance

Micro- and Macro-Ops: What Are They?

- happens in decode stage
- completely hidden from user
- can be split and combined in different stages
- both increase performance

- Micro-Ops
  - splitting 1 instruction
- Macro-Ops
  - combining multiple instructions

Micro- and Macro-Ops: What Are They?

- happens in decode stage
- completely hidden from user
- can be split and combined in different stages
- both increase performance

- Micro-Ops
  - splitting 1 instruction
  - can fetch 1 instruction

- Macro-Ops
  - combining multiple instructions
  - requires you to fetch more than you can decode

Micro- and Macro-Ops: What Are They?

- happens in decode stage
- completely hidden from user
- can be split and combined in different stages
- both increase performance

- Micro-Ops
  - splitting 1 instruction
  - can fetch 1 instruction
  - x86 uses micro-ops **very** often
  - XCHG (R,R/R,M) - 3 operations
- Macro-Ops
  - combining multiple instructions
  - requires you to fetch more than you can decode
  - handled as 1 internal op

A question for the audience

How do Macro-Ops bring High Performance?

A question for the audience

How do Macro-Ops bring High Performance?

They can reduce the 'effective' instruction count.

The Cost of Macro-Op Fusion

- Fetch more than you can decode

The Cost of Macro-Op Fusion

- Fetch more than you can decode
- Search instruction stream for possible matches

The Cost of Macro-Op Fusion

- Fetch more than you can decode
- Search instruction stream for possible matches
- Search before cache-fill

The Cost of Macro-Op Fusion

- Fetch more than you can decode
- Search instruction stream for possible matches
- Search before cache-fill
  - single-issue core: search in cache instead

The Cost of Macro-Op Fusion

- Fetch more than you can decode
- Search instruction stream for possible matches
- Search before cache-fill
  - single-issue core: search in cache instead
- Compiler already does it for you

The Cost of Macro-Op Fusion

- Fetch more than you can decode
- Search instruction stream for possible matches
- Search before cache-fill
  - single-issue core: search in cache instead
- Compiler already does it for you
  - to what extent?

The Cost of Macro-Op Fusion

- Fetch more than you can decode
- Search instruction stream for possible matches
- Search before cache-fill
  - single-issue core: search in cache instead
- Compiler already does it for you
  - to what extent?
- RVC + Macro-Op fusion - CISC behaviour?

Microtraps

Microtraps

- Software emulation routine

Microtraps

- Software emulation routine
- Decoder identifies presence of microtrap

Microtraps

- Software emulation routine
- Decoder identifies presence of microtrap
- PC points to sw routine

Microtraps

- Software emulation routine
- Decoder identifies presence of microtrap
- PC points to sw routine
- Invisible traps? - multi cycle, hidden from user

Examples

x86's Load Effective Address (LEA)

Consider the following piece of C code:

```
struct Point
{
int x;
int y;
}

...
A = points[i].y;
B = &points[i].y;
```

- What is the disassembly? [a]

[a] Assume ebx has array base addr, eax has 'i'.

x86's Load Effective Address (LEA)

Consider the following piece of C code:

```
struct Point
{
int x;
int y;
}

...
A = points[i].y;
B = &points[i].y;
```

- What is the disassembly? [a]
  - (A) mov edx, [ebx + 8*eax + 4]
  - (B) lea edx, [ebx + 8*eax + 4]

[a] Assume ebx has array base addr, eax has 'i'.

x86's Load Effective Address (LEA)

Consider the following piece of C code:

```
struct Point
{
int x;
int y;
}

...
A = points[i].y;
B = &points[i].y;
```

- What is the disassembly? [a]
  - (A) mov edx, [ebx + 8*eax + 4]
  - (B) lea edx, [ebx + 8*eax + 4]
- Broken down into:
  - slli rd, rs1, {1, 2, 3}
  - add rd, rd, rs2

[a] Assume ebx has array base addr, eax has 'i'.

Some More Idioms

- Indexed loads: $rd = array[offset]$
  - add rd, rs1, rs2
  - ld rd, rs1, rs2
- Clear upper word: rd = rs1 & 0xffffffff
  - slli rd, rs1, 32
  - srli rd, rd, 32

Some More Idioms

- Indexed loads: $rd = array[offset]$
  - add rd, rs1, rs2
  - ld rd, rs1, rs2
- Fused indexed loads (with *LEA*)
  - slli rd, rs1, 1, 2, 3
  - add rd, rs1, rs2
  - rd, 0(rd)
- Clear upper word: rd = rs1 & 0xffffffff
  - slli rd, rs1, 32
  - srli rd, rd, 32
- Load immidiate: $rd = imm[31:0]$
  - lui rd, imm[31:12]
  - addi rd, rd, imm[11:0]

Some Example Code

```
800010b8: 00001a97          auipc s5,0x1
800010bc: 5b8a8a93          addi s5,s5,1464 # 80002670 <main+0x450>
800010c0: 04000b93          li s7,64
800010c4: 00391793          slli a5,s2,0x3
800010c8: 00f407b3          add a5,s0,a5
800010cc: 000cb683          ld a3,0(s9)
800010d0: 0007b703          ld a4,0(a5)
800010d4: 000d0613          mv a2,s10
800010d8: 000a8513          mv a0,s5
800010dc: 00d76733          or a4,a4,a3
800010e0: 00e7b023          sd a4,0(a5)
800010e4: 008cb583          ld a1,8(s9)
800010e8: 0019091b          addiw s2,s2,1
800010ec: 008d0b13          addi s6,s10,8
800010f0: 3c5000ef          jal ra,80001d04 <printf>
800010f4: 00391793          slli a5,s2,0x3
800010f8: 00f407b3          add a5,s0,a5
800010fc: 008cb683          ld a3,8(s9)
80001100: 0007b703          ld a4,0(a5)
80001104: 000b0613          mv a2,s6
80001108: 000a8513          mv a0,s5
8000110c: 00d76733          or a4,a4,a3
80001110: 00e7b023          sd a4,0(a5)
80001114: 008cb583          ld a1,8(s9)
80001118: 0019091b          addiw s2,s2,1
8000111c: 008c8c13          addi s8,s9,8
80001120: 595000ef          jal ra,80001eb4 <printf>
80001124: 00391793          slli a5,s2,0x3
80001128: 00f407b3          add a5,s0,a5
8000112c: 010cb683          ld a3,16(s9)
80001130: 0007b703          ld a4,0(a5)
80001134: 010d0613          addi a2,s10,16
```

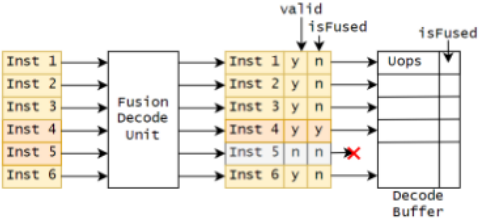
Some Example Code

```
800010b8: 00001a97          auipc s5,0x1
800010bc: 5b8a8a93          addi s5,s5,1464 # 80002670 <main+0x450>
800010c0: 04000b93          li s7,64
800010c4: 00391793          slli a5,s2,0x3
800010c8: 00f407b3          add a5,s0,a5
800010cc: 000cb683          ld a3,0(s9)
800010d0: 0007b703          ld a4,0(a5)
800010d4: 000d0613          mv a2,s10
800010d8: 000a8513          mv a0,s5
800010dc: 00d76733          or a4,a4,a3
800010e0: 00e7b023          sd a4,0(a5)
800010e4: 008cb583          ld a1,8(s9)
800010e8: 0019091b          addiw s2,s2,1
800010ec: 008d0b13          addi s6,s10,8
800010f0: 3c5000ef          jal ra,80001d04 <printf>
800010f4: 00391793          slli a5,s2,0x3
800010f8: 00f407b3          add a5,s0,a5
800010fc: 008cb683          ld a3,8(s9)
80001100: 0007b703          ld a4,0(a5)
80001104: 000b0613          mv a2,s6
80001108: 000a8513          mv a0,s5
8000110c: 00d76733          or a4,a4,a3
80001110: 00e7b023          sd a4,0(a5)
80001114: 008cb583          ld a1,8(s9)
80001118: 0019091b          addiw s2,s2,1
8000111c: 008c8c13          addi s8,s9,8
80001120: 595000ef          jal ra,80001eb4 <printf>
80001124: 00391793          slli a5,s2,0x3
80001128: 00f407b3          add a5,s0,a5
8000112c: 010cb683          ld a3,16(s9)
80001130: 0007b703          ld a4,0(a5)
80001134: 010d0613          addi a2,s10,16
```

Fusion Decoder Unit

XiangShan Processor

- up to six instructions from instruction buffer
- set valid to false if fused
- cannot enter decode buffer



Fusion Decoder Unit in XiangShan Processor

Fusion Targets

Target	Instruction Pairs	Description
szewl1	slli r1, r0, 32 + srli r1, r1, 31	left shift zero-extended word by 1 bit
szewl2	slli r1, r0, 32 + srli r1, r1 30	left shift zero-extended word by 2 bits
szewl3	slli r1, r0, 32 + srli r1, r1 29	left shift zero-extended word by 3 bits
sh4add	slli r1, r0, 4 + add r1, r1, r2	left shift by 4 bits and add
sr29add	srli r1, r0, 29 + add r1, r1, r2	right shift by 29 bits and add
sr30add	srli r1, r0, 30 + add r1, r1, r2	right shift by 30 bits and add
sr31add	srli r1, r0, 31 + add r1, r1, r2	right shift by 31 bits and add
sr32add	srli r1, r0, 32 + add r1, r1, r2	right shift by 32 bits and add
oddaddw	andi r1, r0, 1 + addw r1, r1, r2	add one if odd (in word format)
orh48	andi r1, r0, -256 + or r1, r1, r2	or operation with high 48 bits

Evaluation Results of Coremark

feature	commited instr	cycle count	coremark/MHz
plain	36,833,874	15,108,897	6.70
fusion	34,999,666	14,840,832	6.82
“B” extension	31,220,576	13,563,862	7.47
“B” extension + fusion	30,151,534	13,485,735	7.51

Open for Discussions

Thank You

Follow up question

What if the front-end fuses two instructions together to save decode, allocation, and commit bandwidth, but breaks them apart in the execution pipeline for critical path or complexity reasons?