J1: Compilers Challenges for Computing In Memory

Micro architecture & Compilers Internals

Henri-Pierre CHARLES

CEA DSCIN department / Grenoble

july 11, 2022

Intro : Presentation Bridge

Where are the missing parts ?

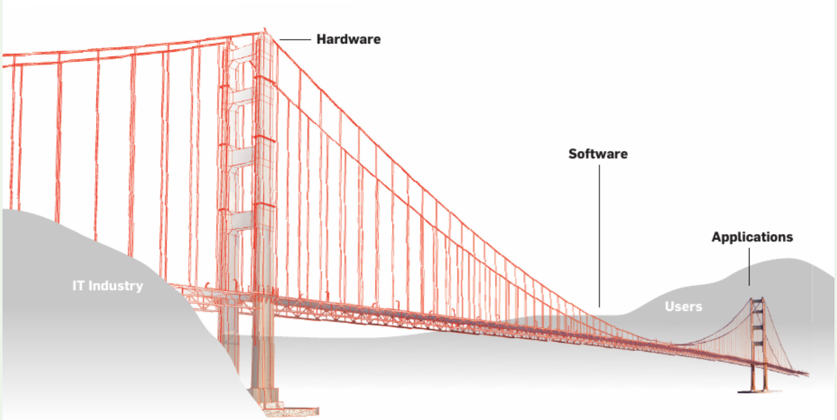


Figure 2. Bridge analogy connecting users to a parallel IT industry, inspired by the view of the Golden Gate Bridge from Berkeley, CA.

[Ref]K. Asanovic et al., "A View of the Parallel Computing Landscape"

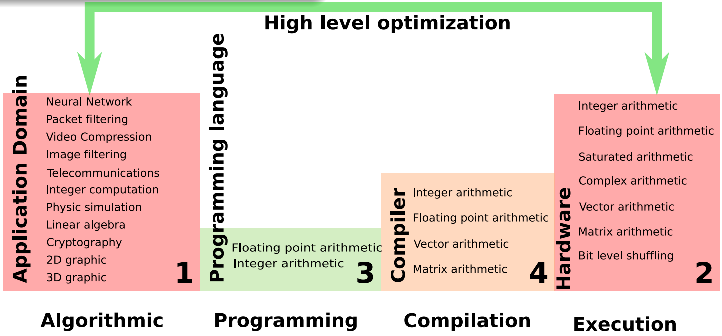
Introduction : What's the problem ?

Programming language (When are you born ? :-)

- Are invented and **standardized** between 70 and 90'
- Based on logic, integer and floating point arithmetic.
- Compiler use arithmetic rules for optimization

Computer architecture

- Use real world datasets : bitmap images, vector images, datagram, ...
- Hardware development and Low level programming research domains are deconnected



Introduction : C Data Types

Basic Data representation

Integers W Binary-coded_decimal, W Two's_complement (C)

Characters W Baudot_code (1901), W ASCII (1970) (C), W Latin 1 1998, W UTF-8 1992,

Floating point Intel, Ibm, IEEE (C), Unum, Posit, VRP, Stochastic

Others : Pixels (RGV, YUV, CMJN, ...), IP addresses, IA Objects (Cats, dogs, ...)

Constructed Data representation

Integer array index

Characters Strings, Text

Chaînes de caractères Chaînes, Texte

Floating point matrix, vector

Pixels Images (array of struct, struct of array), Sprites,

TCP packet ../.. (look at BPF : compilation for packet filtering)

Type Based Compiler Optimisations

Integer Algebra rules : commutativité, distributivité

Character None

Floating point None, unless using `-fast-math` which broke algorithms based on numerical stability

Pixels None

IP addresses None (BPF DSL specialized compiler)

IA objects Human slaves

Constructed Data Representation

- Compiler has not clue of the semantic of the constructed datatype.
- "Leave the axe to the programmer" !
- Please give complex data type semantic to a compiler

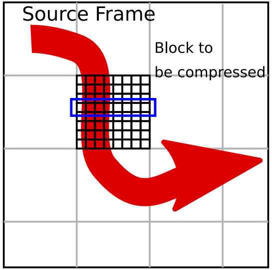
Introduction : Problem Example Image Compression

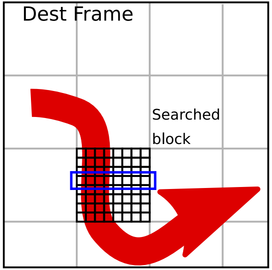
ARM specialized instruction

- `usad8 rd, rs1, rs2` (Page 4482 / 7476)
- Used in all video compression tools (VLC, FFMPEG)
- [Ref]A Global Approach For MPEG-4 AVC Encoder Optimization

Image compression basic : block comparison

- $Res = \sum_{i=0}^{8} |in0[i] - in1[i]|$
- No compiler support, should use assembly level programming
- (See Videolan VLC project)
- Software support :
  - Assembly level function : no compiler support
  - Programmer tiling model : no compiler support





Introduction : VLC Implementation

https://github.com/vlc-mirror/

https://github.com/vlc-mirror/x264/blob/master/common/arm/pixel-a.S

```c
/****************************************************************
 * pixel_sad_WxH
 ****************************************************************
#define PIXEL_SAD_C( name, lx, ly ) \
static int name( pixel *pix1, intptr_t i_stride_pix1,  \
                 pixel *pix2, intptr_t i_stride_pix2 ) \
{                                                       \
    int i_sum = 0;                                      \
    for( int y = 0; y < ly; y++ )                       \
    {                                                   \
        for( int x = 0; x < lx; x++ )                   \
        {                                               \
            i_sum += abs( pix1[x] - pix2[x] );          \
        }                                               \
        pix1 += i_stride_pix1;                          \
        pix2 += i_stride_pix2;                          \
    }                                                   \
    return i_sum;                                       \
}


PIXEL_SAD_C( x264_pixel_sad_16x16, 16, 16 )
PIXEL_SAD_C( x264_pixel_sad_16x8,  16,  8 )
```

https://github.com/vlc-mirror/x264/blob/master/common/pixel.c

```asm
.macro SAD4_ARMV6 h
function x264_pixel_sad_4x\h\()_armv6
    push        {r4-r6,lr}
    ldr         r4, [r2], r3
    ldr         r5, [r0], r1
    ldr         r6, [r2], r3
    ldr         lr, [r0], r1
    usad8       ip, r4, r5
.rept (\h - 2)/2
    ldr         r4, [r2], r3
    ldr         r5, [r0], r1
    usada8      ip, r6, lr, ip
    ldr         r6, [r2], r3
    ldr         lr, [r0], r1
    usada8      ip, r4, r5, ip
.endr
    usada8      r0, r6, lr, ip
    pop         {r4-r6,pc}
```

Models : C Language for Architecture

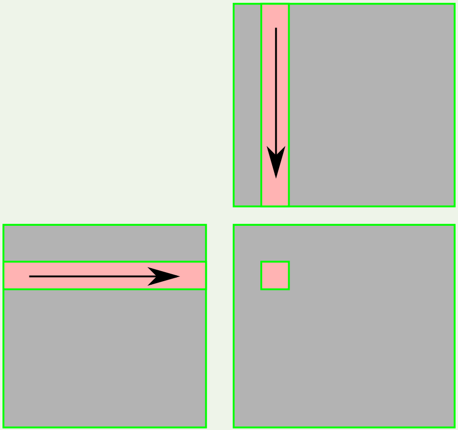
Matrix multiply (sketch)

```
for (int l = 0; l < SIZE; l++)
  for (int c = 0; c < SIZE; c++)
    for (int k = 0; k < SIZE; k++)
      R[l][c] += A[l][k] * B[k][c];
```

"Real world"

```
for (c= 0; c<NCOL; c+=cacheLineSize)
 for (l= 0; l<NLINE; l+=halfCacheLine)
  for (c2= 0; c2<NCOL; c2+=halfCacheLine)
    for (lk= 0; lk<halfCacheLine; lk++)
     for (c2k= 0; c2k<halfCacheLine; c2k++)
      for (ck= 0; ck<cacheLineSize; ck++)
        res[l+lk][c2+c2k]+=     a[l+lk] [c+ck]* b[c2+c2k][c+ck];
```

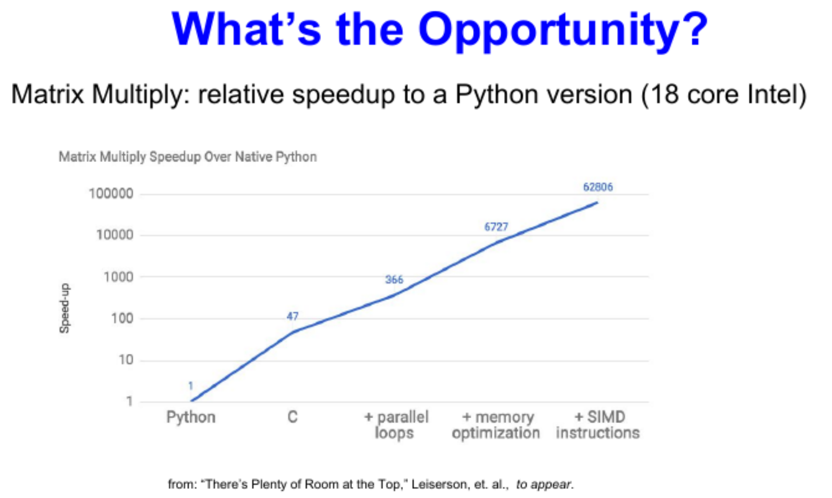
Learn to program = learn to serialize / schedule on defined hardware !



Other "interesting examples"

Introduction : plenty room at the top

Code optimization



Intro : Compilation

Source

```c
#include <stdio.h>

int main(int arc, char * arrgv[])
{
  printf("Hello world !\n");
  return 0;
}
```

Un **compilateur** est un logiciel permettant de transformer **un programme source** (écrit dans un langage de programmation) dans un autre langage **de programmation cible**, le plus souvent dans le **langage d'un processeur** permettant **d'exécuter** le dit programme.

Destination

```
000000000000063a <main>:
 63a:    55
 63b:    48 89 e5
 63e:    48 83 ec 10
 642:    89 7d fc
 645:    48 89 75 f0
 649:    48 8d 3d 94 00 00 00
 650:    e8 bb fe ff ff
 655:    b8 00 00 00 00
 65a:    c9
 65b:    c3
 65c:    0f 1f 40 00
```

Intro : Compilation

Technology which link

- Link between artist and machine
- Expressiveness
- Automatism
- Numeric to analog
- Huge economic effect
- Huge social effect

Joseph Marie Charles dit "Jacquard"

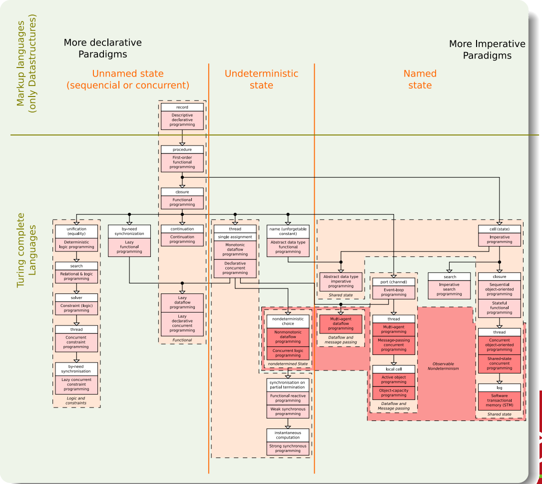
Introduction : Programming Model Paradigms

What's a programming model ?

- A Knowledge in the programmer mind (aka W Programming_paradigm):
  - imperative procedural
  - object-oriented
  - declarative
  - functional
  - logic
  - mathematical ../..
- A real language which as multiple paradigms : W Comparison_of_multi-paradigm_programming_languages

Wich paradigm link to hardware features ?

None



Scientific Evolution : Compilation Research Domains

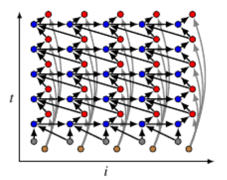
Compilation Topics Map

- **Find code structure** Extract parallelism : polyhedral approach
- **Assertion on legacy code** correctness proof, hard realtime, model checking
- **Security** HW attach counter mesure, obfuscation
- **Tools for scalability** Systems and library for big parallel machines
- **Reproductibility** Statistics tools and reproductible research
- **Ad hoc code optimization** Application driven code optimization
- **Legacy compiler optimization** follow the HW evolution
- **New code generation paradigm** JIT, Dynamic code generation

https://top500.org

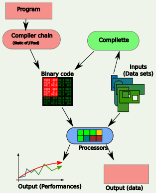
- **LAPACK** June 2002 #1 : HPE Cray 8,730,112 cores, Linpack perf 1,102.00 PFlop/s, 65% of peak performance (usualy better)
- **HPCG** Fugaku 7630848 cores, 16,004 TFlops 2.98 % of peak performance !

Polyhedral model



(a) CDAG for $T = 4, N = 7$

Algorithmic accelerator



CPU Programming Model (MC68000)

What's inside the programmer head ?

- Not only the ISA + addressing modes
- "Some" processors notions :
  - Registers
  - UALs / vectors width / # parallel UAL
  - Pipeline
  - Branch penalty
  - Caches L1 $D $I / Caches L2 / Caches L3
  - Interprocessors communication bandwith versus Latency
  - ...

MC 68000 "Programming model"

Freescale Semiconductor, Inc.

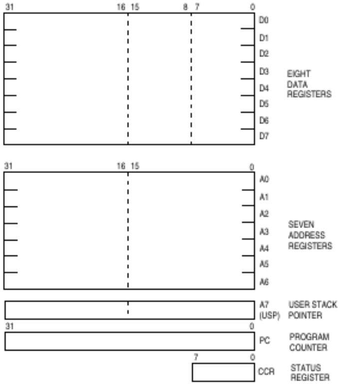


Figure 2-1. User Programmer's Model (MC68000/MC68HC000/MC68008/MC68010)

Introduction : Low Level ProgrammingModel

ISA is the frontier between HW & SW

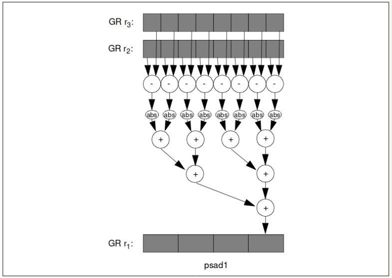
- Give parsable semantic (even if not "arithmetic compatible")
- Give programmer comprehension
- Give hardware semantic
- Example : https://www.intel.com/content/dam/www/public/us/en/architecture-vol-3-manual.pdf

psad — Parallel Sum of Absolute Difference

Format: (qp) psad1 $r_1 = r_2, r_3$ I2

Description: The unsigned 8-bit elements of GR r_2 are subtracted from the unsigned 8-bit elements of GR r_3. The absolute value of each difference is accumulated across the elements and placed in GR r_1.

Figure 2-38. Parallel Sum of Absolute Difference Example



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
    x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
    x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
    x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
    x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
    x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
    x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
    x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};

    GR[r1] = 0;
    for (i = 0; i < 8; i++) {
        temp[i] = zero_ext(x[i], 8) - zero_ext(y[i], 8);
        if (temp[i] < 0)
            temp[i] = -temp[i];
```

SOA Arch : Language Classification / Addresses

Definition

- 0 address : stack machines; all operators are implicit (bytecode java) (Java bytecode, postscript)
- Register machines :
  - 1 address : A add (other operand implicits, ARM/Jazelle)
  - 2 address : A += B (x86, ARM thumb)
  - 3 address : A = B + C (itanium, ARM, RISCV, power, kalray)
  - 4 address : A = B * C + D (power)

W Comparison_of_instruction_set_architectures

Compromise

- Code compacity, expressiveness
- "Simplicity" / efficiency
- Code generation speed

HWParallelismLevel uArch

What every programmer should know about performance

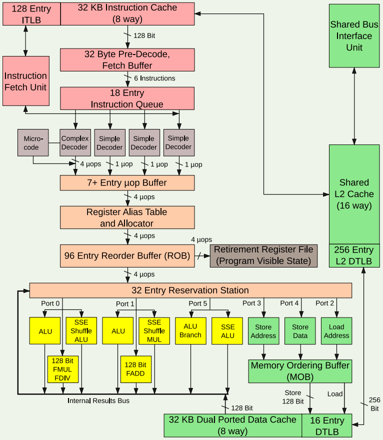
- Hidden micro architecture
- Memory hierarchy

Intel Example

- "700" simple high level instructions
- 17000+ instructions variants
- RISC internal uInstructions

Intel Micro architecture

Intel Core2 micro arch



Intel Core 2 Architecture

SOA ARCH : ARM big.LITTLE

big

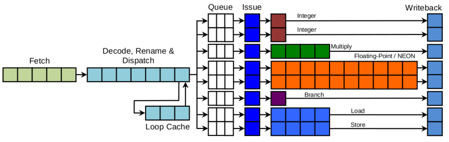


Figure 2 Cortex-A15 Pipeline

LITTLE

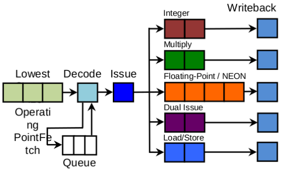


Figure 1 Cortex-A7 Pipeline

ARM : "more than 30 billion processors sold with more than 16M sold every day ARM" (Nov 2013) `http://www.arm.com/products/processors/index.php`

- 4 big processors + 4 little
- Same ISA, ...
- (even for vector operation)
- Low latencie switch

big.LITTLE notion

HWParallelismLevel GPU

Characteristics

- Heterogeneous processing . In terms of
  - processing CPU + GPU
  - programming language
  - parallelism level
- Need multiple programming language (X + Y)
  $X \in C, Python, \ldots\ Y \in CUDA, OpenCL, \ldots$

How to use

- Organize "data choregraphy"
- Thread synchronization

HWParallelismLevel MPSoC

MPSoC for smart phone

- Aggregate muliple IP on a common die
  - Computing node
  - GPU (+TPU ?)
  - DSP
  - Modem, bluetooth, ...

How to use

- Java for application processor
- Android for OS (with Google store)
- C for DSP, modem

HWParallelismLevel MultiCPU

Multiple CPU, same processor

- Share L3, global memory
- Same peripherals
- Multiple clocks

Best application

- Same peripherical access (disk, network if)
- Big data parallel applications

Arch : Simulation Platforms and Levels

Argumentation

- Atomistic level / Physicists
- Electronic level / HW engineer
- Instruction level / Compiler & application
- System level /

Which level is our science ?

Name	In	Out	Time
W BigDFT	Atoms position	Electonic behavior	ns
W SPICE	Elect. behavior + layout	ms	
W VHDL	Bloc behavior		
W SystemC	System behavior	Event	
W Gem5	Binary code	Functionnal behavior	few sec.
W QEMU	Binary code	Functionnal behavior	real time

HWParallelismLevel VLIW

Argumentation

- Explicit use of multiple ALU
- Example for MM
  - 2 add for address computation
  - 2 load from memory
  - 1 MAD
  - 1 store

Example

- Intel Itanium (2 bundle of 3 parallel instructions)
- Kalray MPPA (See Kalray presentation)

Metric

- Insn / cycle
- Bundle filling rate

HWParallelismLevel Cluster

Multiple systems

- Share network
- Multiple clocks
- Multiple disks
- Multiple OS
- Multiple languages

Illustration

- Multiple tier applications
- Web applications (LAMP or variants)
- Complex paralle data analysis

HWParallelismLevel ILP

ILP - Instruction Level Parallelism

- How to use multiple ALU
- How to use special instructions. Examples
  - Multiply and Add (matrix multiply)
  - SAD (Video compression)

Metrics

- Count clock cycle / instruction count
- Compute INSN / cycle
- Verify against CPU description

HWParallelismLevel MultiCore

Many computing core

- Multiple instruction flow
- Asynchronous programming
- Share the same die

How to use

- Same die : should optimize data / code sharing
- Augment spacial data locality
- Augment code sharing

Best usage : Same code / data

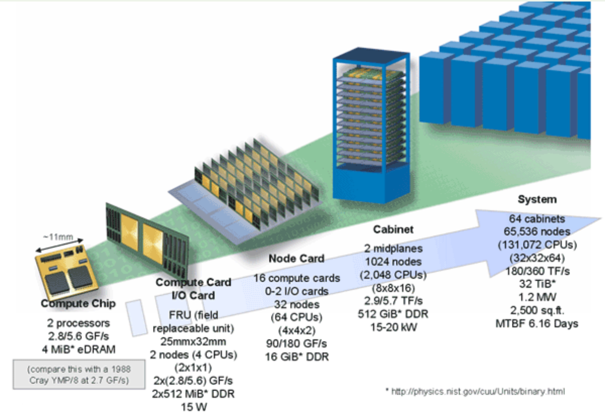
- Numeric simulation
-

HWParallelismLevel MultipleLevel

Argumentation

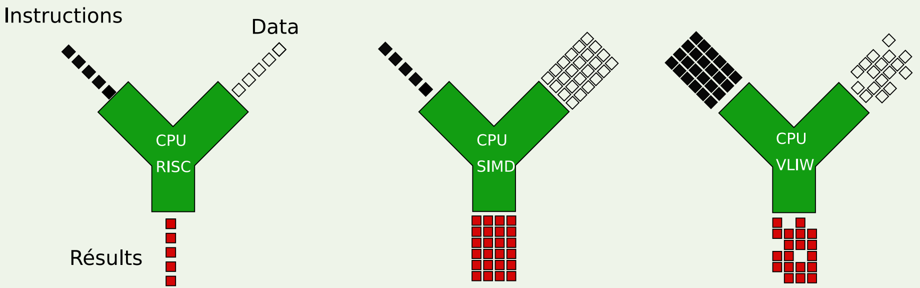
- Which level fit ?
  - ALU, Operator
  - VLIW, OOO, Instruction, uArch
  - Multicore
  - MultiCPU
  - Multi computer
  - Heterogeneity

Illustration : Blue Gene Machine



Processor Words and Instructions Size

How to deal with low level parallelism



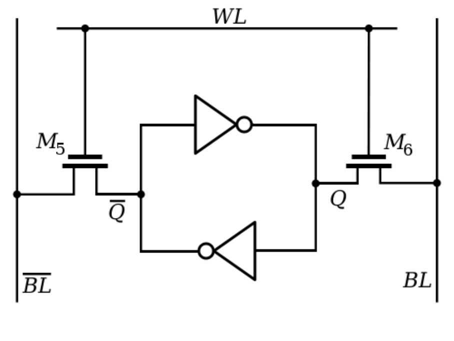
- Scalar RISC CPU : "simple" for compiler
- SIMD CPU aka Vector instructions : good for dense computation
- VLIW CPU : good for complex workload

Introduction : Remember Memory Cell 101

SRAM memory cell depicting Inverter Loop as gates

- 6T memory cell
- Only 1 stable mode
- Read : "open" `WL`, read value
- Write : "open" `WL`, write value

Illustration



W Memory_cell_(computing)

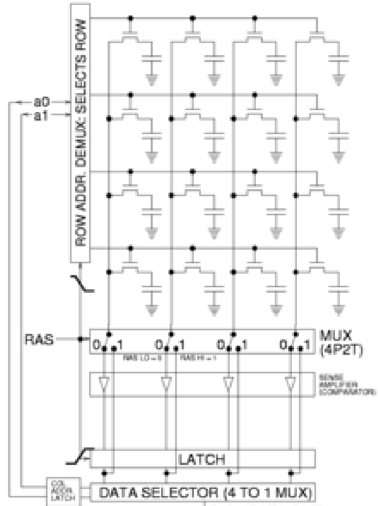
Introduction : Remember Memory 102

Functions

- Select line
- Read or write
- Potentially select word in a line
- Low voltage used; "Sense amp" to normalize

W Sense_amplifier
What every programmer should know about memory

Memory array



Intro : CSRAM Memory

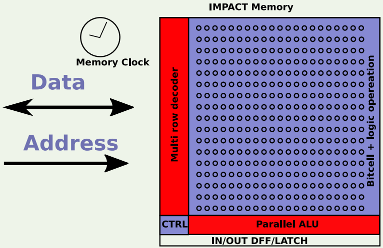
Memory with

- Bitcell 2 ports
- Vector like alu
- Multi-line selector
- No internal instruction execution

Design choices

- Minimize CPU modification
- Only one instruction sequence (CPU master)

IMPACT memory



Technology

SRAM technology (FDSOI 22nm)

Arch : Risc Versus Cisc

Difference between CISC & RISC

- CISC : complex instruction set computer
- RISC : reduced instrution set computer
- Is it no more "simple" versus "complex" !
- Load-store architecture

Instruction level Data access

<table>
<tr><th></th><th>ALU</th><th>Reg. bank</th><th>Caches</th><th>Memory</th></tr>
<tr><td>RISC</td><td colspan="2"><Arithmetic instructions></td><td></td><td></td></tr>
<tr><td>RISC</td><td></td><td colspan="3"><Data access instructions></td></tr>
<tr><td>CISC</td><td colspan="4"><All instructions access></td></tr>
</table>

Arch : RISC-V

RISCV

- Open source Instruction set `https://riscv.org/technical/specifications/`
- Open source implementations
- Foundation based model (Swiss)
- Enterprises based on RISCV : SiFIVE, SiPEARL, GreenWavesTech..
- Univesity based on RISCV : ETHZ `https://pulp-platform.org/`
- RISC instruction set
-

Instruction extensions

Bases : RVWMO, RV32I Base Integer, RV32E Embedded, RV64I, RV128I
Many extensions RISC-V\#ISA_base_and_extensions:

- M: Multiplication
- A: Atomics – LR/SC & fetch-and-op
- F: Floating point (32-bit)
- D: FP Double (64-bit)
- Q: FP Quad (128-bit)
- Zicsr: Control and status register support
- C: Compressed instructions(16-bit)
- J: Interpreted or JIT compiled languages support
- G : Shorthand for the IMAFDZicsr Zifencei base and extensions, intended to represent a standard general-purpose ISA
- V : Standard Extension for Vector Operations
- P : Standard Extension for Packed-SIMD Instructions
- & many others

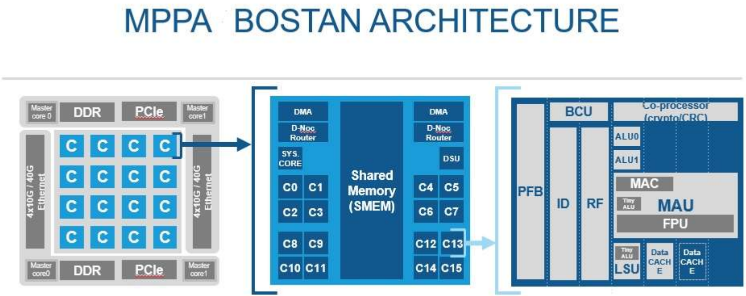
Arch : Kalray

Argumentation

- Multiple grid of cores (MPSoC)
- Scratchpad Memory
- VLIW instruction set (explicit parallelism)
- Not yet open instruction set description
- 256 cores
- 1234 page ISA description

Parameters

- 5 issues architecture
- 64 registers
- 64 bits



Arch : ARM-ISA

ARM_architecture_family

- ARM-A / Architecture AArch64 ISA A64 / AArch32 A32 / T32 "Application"
- ARM-R Cortex-R ARM_Cortex-R "Real time"
- ARM-M Cortex-M ARM_Cortex-M "eMbedded" (16 bits ISA) .. less registers
- 7000+ page isa documentation

Instruction set familly

"classical" 32 bits instruction set (load/store)

Thumb Compressed instruction set

Jazelle Java support

Neon Advanced SIMD

ISA : Instruction Encoding

Instruction Encoding Trade-Off

- How to encode a data range access ?
- How to encode a large parallelism ?
- How to provide “easy” SW acessibility ?

Examples

- How many registers ? 64 registers need 6 bits, instruction with 3 addresses = 18 bits
- Data word width ? Compromise between numérical / vector len and bus width (energy !)
- How many parallel operation ?
  - Vector
  - VLIW

Arch : Instruction Format

ISA

- How many instructions ? opcode width
- How many registers ? register encoding width
- Data size ? immediated encoding width
- Example on 32 bits RISCV instructions :

<table>
<tr><th rowspan="2">Format</th><th colspan="32">Bit</th></tr>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>Register/register</td><td colspan="7">funct7</td><td colspan="5">rs2</td><td colspan="5">rs1</td><td colspan="3">funct3</td><td colspan="5">rd</td><td colspan="7">opcode</td></tr>
<tr><td>Immediate</td><td colspan="12">imm[11:0]</td><td colspan="5">rs1</td><td colspan="3">funct3</td><td colspan="5">rd</td><td colspan="7">opcode</td></tr>
<tr><td>Upper immediate</td><td colspan="20">imm[31:12]</td><td colspan="5">rd</td><td colspan="7">opcode</td></tr>
<tr><td>Store</td><td colspan="7">imm[11:5]</td><td colspan="5">rs2</td><td colspan="5">rs1</td><td colspan="3">funct3</td><td colspan="5">imm[4:0]</td><td colspan="7">opcode</td></tr>
<tr><td>Branch</td><td>[12]</td><td colspan="6">imm[10:5]</td><td colspan="5">rs2</td><td colspan="5">rs1</td><td colspan="3">funct3</td><td colspan="4">imm[4:1]</td><td>[11]</td><td colspan="7">opcode</td></tr>
<tr><td>Jump</td><td>[20]</td><td colspan="10">imm[10:1]</td><td>[11]</td><td colspan="8">imm[19:12]</td><td colspan="5">rd</td><td colspan="7">opcode</td></tr>
</table>

- *opcode* (7 bits): Partially specifies which of the 6 types of *instruction formats*.
- *funct7*, and *funct3* (10 bits): These two fields, further than the *opcode* field, specify the operation to be performed.
- *rs1*, *rs2*, or *rd* (5 bits): Specifies, by index, the register, resp., containing the first operand (i.e., source register), second operand, and destination register to which the computation result will be directed.