SimCore/Alpha Functional Simulator Version 1.0 : Simple and Readable Alpha Processor Simulator

Kenji KISE *kis@is.uec.ac.jp*

Graduate School of Information Systems,
University of Electro-Communications

Outline

- Introduction
- SimCore/Alpha overview
- SimCore/Alpha internals
- SimCore/Alpha practical use
- Summary

Introduction

- Various processor simulators are used for research and education activities.
- Famous **SimpleScalar Tool Set** is not a code that can easily be modified.

- **We have developed** a processor simulator **SimCore/Alpha Functional Simulator Version 1.0** (**SimCore/Alpha Version 1.0** in short) .
- Its design policy is to keep the source code **readable** and **simple**.
- Software architecture of **SimCore/Alpha** is explained by referring to its source code.
- As **SimCore/Alpha** practical use, we present the ideal instruction-level parallelism of SPEC benchmarks measured with a modified version of **SimCore/Alpha**.

SimCore/Alpha overview

This section overviews the SimCore/Alpha.
We focus the design policy, simulation plathomes, execution image file format and simulation speed.

SimCore/Alpha Version 1.0 design policy

- **Target applications**
  - SPEC CINT95 and CINT2000.
- **Simple and readable (enjoyable and easy to read) code**
  - No global variable
  - No goto statement
  - No conditional compilation
  - The source code is only about 2,800 lines in C++.
- **It offers another choice**
  - A processor simulator is an important tool.
  - It is advantageous to choose a suitable tool from many choices.
  - SimCore/Alpha offers another choice.
- **Functional simulator, but**
  - Although only the capability of a functional simulator is given, the code is written considering the extension to the out-of-order processor simulator.

SimCore/Alpha Version 1.0 is a functional simulator.

- Functionally equivalent to the **sim-safe** or **sim-fast** of **SimpleScalar Tool Set**.
- Functional Simulator is available for
  - Simulator verification
  - Evaluation of instruction mix
  - Evaluation of branch predictors
  - Evaluation of memory systems
  - Evaluation of ideal instruction-level parallelism.

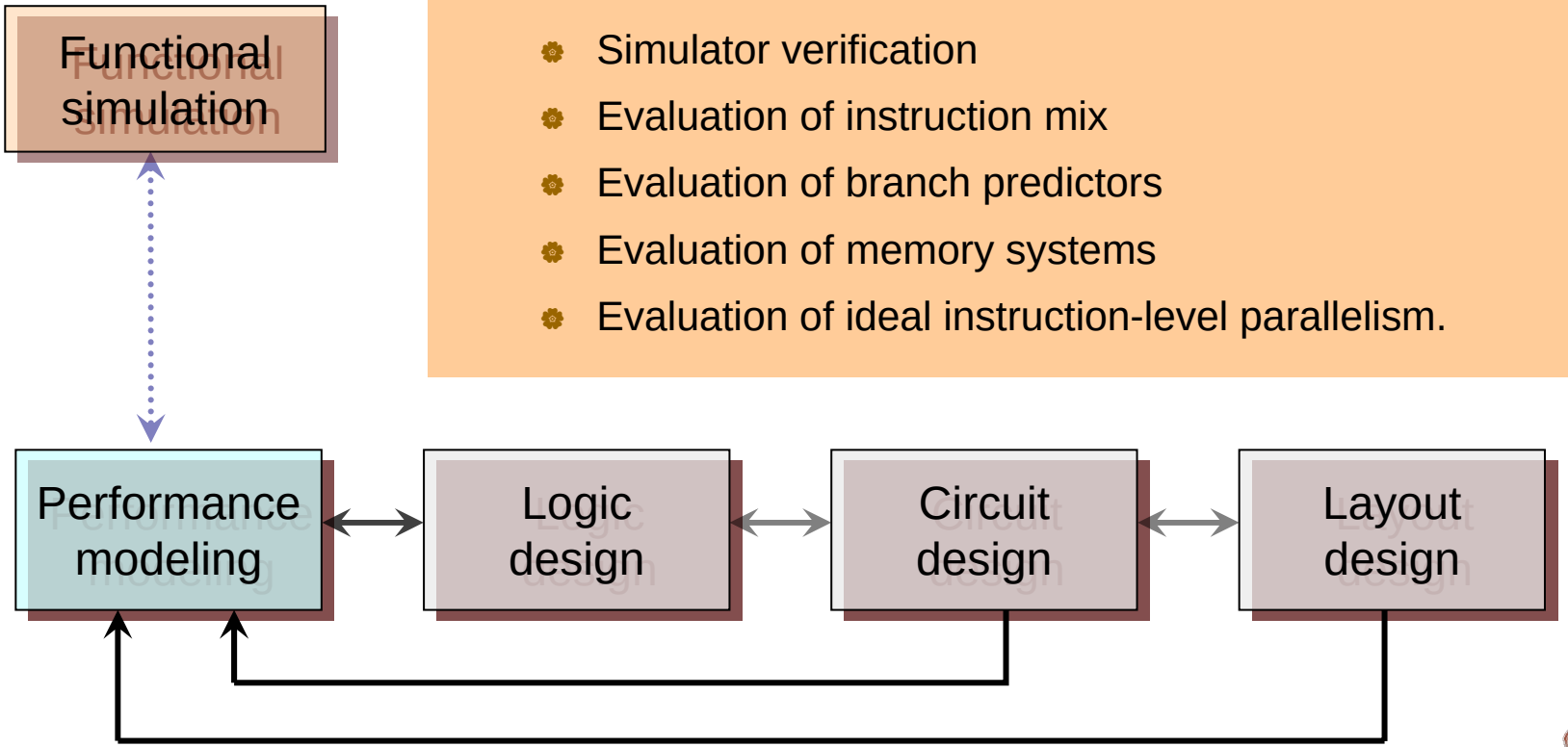


Figure 1. Typical flow in microprocessor design process. Interaction between the process steps refines the performance model throughout the process. (IEEE Computer, February 2002, p 30)

SimCore/Alpha Version 1.0 software components

```
 Line     Word     Byte   Filename

  340     2968    17992   COPYING
   73      193     1556   Makefile
  356     1570    13174   README.txt
  871     2948    21347   arithmetic.cc
   73      199     1882   chip.cc
  201      626     5286   debug.cc
  368     1173    11142   define.h
  271      811     7751   etc.cc
  187      647     4505   instruction.cc
  179      491     4412   memory.cc
   15       49      417   sim.cc
  562     1234    14871   syscall.cc

 3496    12909   104335   total
 2727     8178    71613   total *.cc *.h
```

- **Licensing**
  GNU General Public License Version 2
- The source code and include file consists of **only 2,727 lines**.
- arithmetic.cc : 871 lines for arithmetic and logic etc.
- syscall.cc : 562 lines for system call implementation.
- **Core source code is about 1,300 lines.**

SimCore/Alpha plathomes

- **SimCore/Alpha Version 1.0** plathomes:
  - Intel Pentium 4, RedHat Linux 7.3, gcc version 2.96
  - Intel Pentium III, RedHat Linux 6.2, gcc
- **SimCore/Alpha Version 1.4** plathomes:
  - Intel Pentium 4, RedHat Linux 7.3, gcc version 2.96
  - Intel Pentium 4, Windows XP Cygwin version 2.340, gcc
  - DEC Alpha 21264, Tru64 UNIX V5.1, gcc version 2.95.2
  - **AMD Opteron, Turbo Linux 8, gcc version 3.2.2**
  - **Intel Pentium 4, RedHat Linux 7.3, Intel C++ Compiler 7.1**

Note that SimCore/Alpha uses no conditional compilation.

SimCore/Alpha verification

- In the development phase of **SimCore/Alpha**, compatibility with SimpleScalar was carefully confirmed.
- When **SimCore/Alpha** executed one instruction, we confirmed that the two architecture states (a program counter, 32 integer registers, 32 floating point registers) were identical.

- **SimCore/Alpha** uses no global variable. Two or more simulation images can easily be generated in one process.
  - Any bug of simulators under development is discovered at an early stage.

Execution image file as SimCore/Alpha Input

```
/* SimCore 1.0 Image File */
/*** Registers ***/
/@reg 16 0000000000000003
/@reg 17 000000011ff97008
/@reg 29 0000000140023e90
/@reg 30 000000011ff97000
/@pc  32 0000000120007d80
/*** Memory    ***/
@11ff97000 00000003
@11ff97008 1ff97138
@11ff9700c 00000001
```

Sample execution image file.

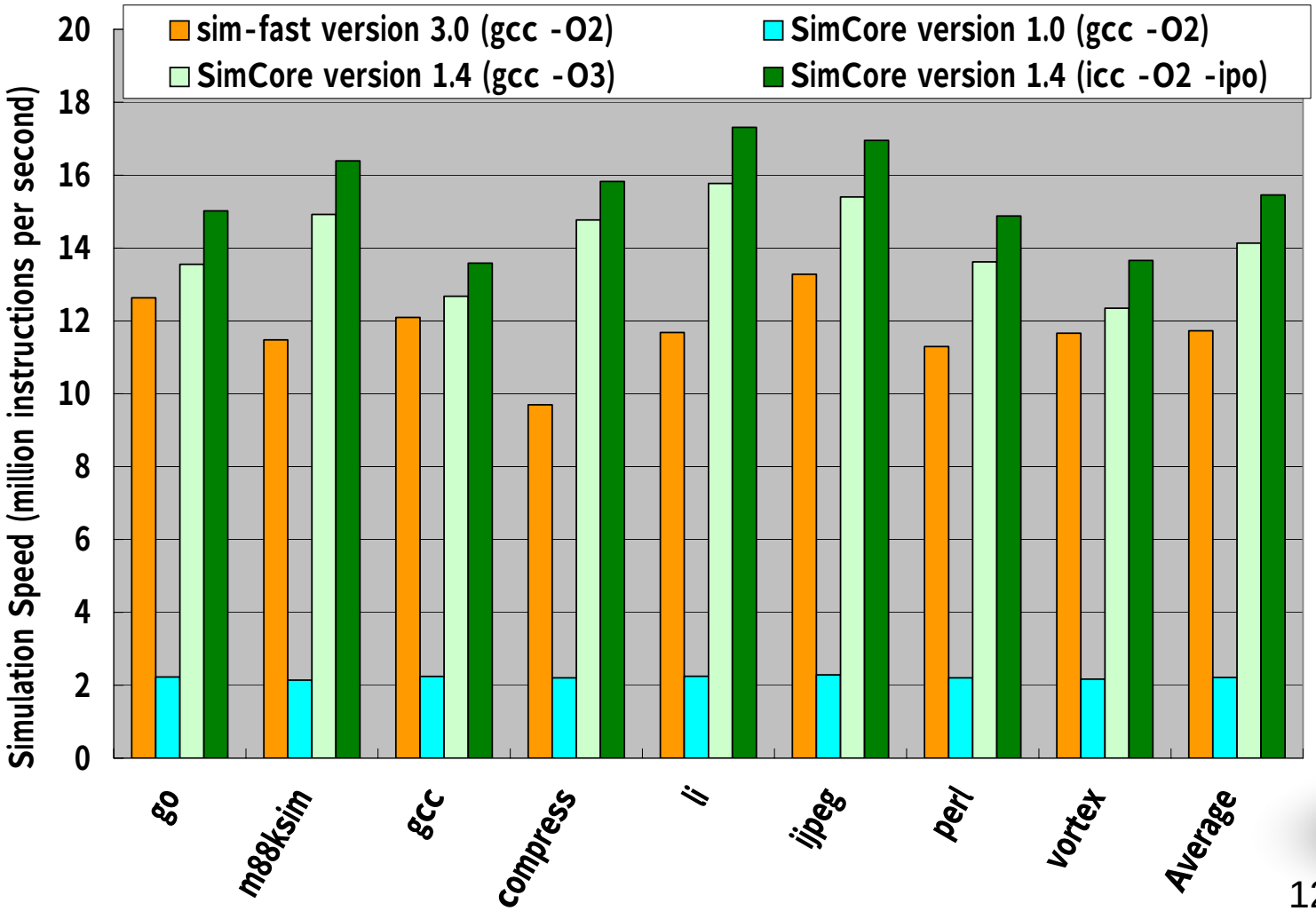
- SimCore/Alpha reads not an Alpha binary but an **execution image file in original format**.
- This simple original format makes the knowledge of executables such as ELF and COFF unnecessary.
- **In the first part**, values are assigned to **the registers**.
- **In the second part**, values are assigned to **the memory**.
- This image file is created from an Alpha binary by **SimCore-Loader**.

SimCore/Alpha simulation speed

- We ran the **8 SPEC CINT95** benchmark programs and calculated the average simulation speed.
- **Pentium 4 Xeon 2.8GHz dual processor PC** with 4GB memory running Red Hat Linux 7.3.
- **Configurations:**
  - **sim-fast** from SimplaScalar Version 3.0c
    - Compiled with GCC with –O2 optimization flag
  - **SimCore/Alpha Version 1.0** : first release of SimCore/Alpha.
    - Compiled with GCC with –O2 optimization flag
  - **SimCore/Alpha Version 1.4** : **optimized version** of SimCore/Alpha.
    - Compiled with **GCC** with **–O3** optimization flag
    - Compiled with **Intel C++ Compiler** with –O2 –ipo optimization flag

SimCore/Alpha simulation speed

- SimCore/Alpha **Version 1.0** is 6 times slower than **sim-fast (SimpleScalar)**.
- **SimCore/Alpha Version 1.4** is **30% faster** than **sim-fast**.
- **Now SimCore/Alpha is simple, readable and fast!**



Fast

SimCore/Alpha internals

In this section, in order to show the **high readability** of the source code, the internal structure of SimCore/Alpha is explained showing actual C++ code (**not pseudo code**).

SimCore/Alpha main function

```
int main(int argc, char **argv){
  if(argc==1) usage();
  char *p    = argv[argc-1]; /* program name */
  char **opt = argv;         /* options      */

  simple_chip *chip =  new simple_chip(p, opt);
  while(chip->step());
  delete chip;

  return 0;
}
```

- Easy to understand and modify.
- No global variable

Sample command

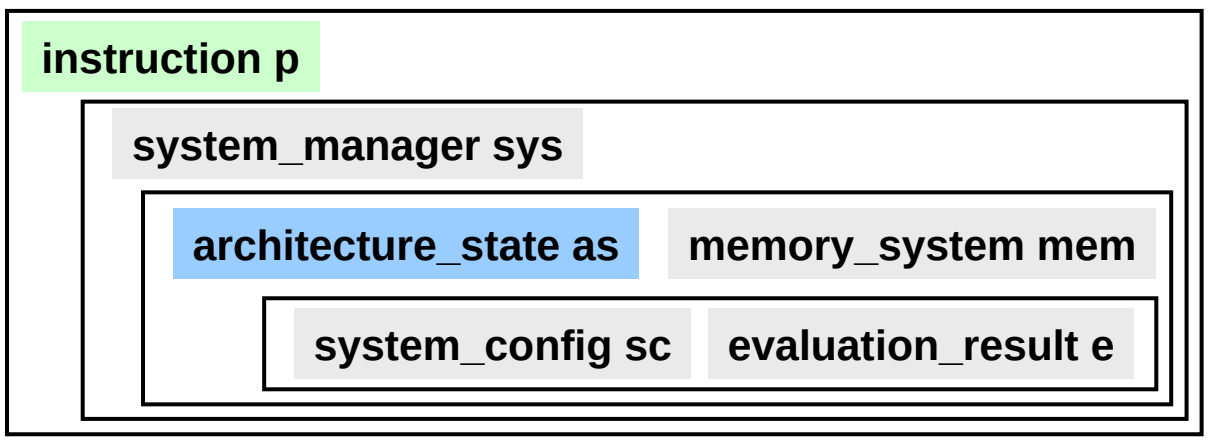
SimCore/Alpha -e10000 -v100 aout.txt

option program name

Class simple_chip constructor

```
simple_chip::simple_chip(char *prog, char **opt){
  sc  = new system_config(prog, opt);
  e   = new evaluation_result;
  as  = new architecture_state(sc, e);
  mem = new memory_system(sc, e);
  deb = new debug(as, mem, sc, e);
  sys = new system_manager(as, mem, sc, e);
  p   = new instruction(as, mem, sys, sc, e);
}
```

simple_chip creates seven objects.



Class data_t definition and methods

```
class data_t{
  uint64_t value;
 public:
  int cmov;
  uint64_t ld();
  int   st(uint64_t);
  int init(uint64_t);
};

int data_t::init(uint64_t d){ value = d; cmov = 0; return 0;}
uint64_t data_t::ld(){ return value; }
int data_t::st(uint64_t d){ value = d; return 0;}
```

- The calculation results are stored in register file or memory.
  These are defined as the collection of **class data_t** objects.
- **Function st** is used to store a data value into a data_t type object.
  Function ld is used to read a data value.
  Function init is used to generate a new object.

Class architecture_state definition

```
class architecture_state{
 public:
  data_t pc;    /* program counter      */
  data_t r[32]; /* general purpose regs */
  data_t f[32]; /* floating point  regs */
  architecture_state(system_config *,
                     evaluation_result *);
};
```

- **The architecture state** of **Alpha-AXP** consists of a program counter, 32 integer registers and 32 floating point registers.
- **The architecture state** is defined as a collection of the data_t type objects.

SimCore/Alpha main function again

```
int main(int argc, char **argv){
  if(argc==1) usage();
  char *p    = argv[argc-1]; /* program name */
  char **opt = argv;         /* options      */

  simple_chip *chip =  new simple_chip(p, opt);
  while(chip->step());
  delete chip;

  return 0;
}
```

- Easy to understand and modify.
- No global variable

Sample command

SimCore -e10000 -v100 aout.txt

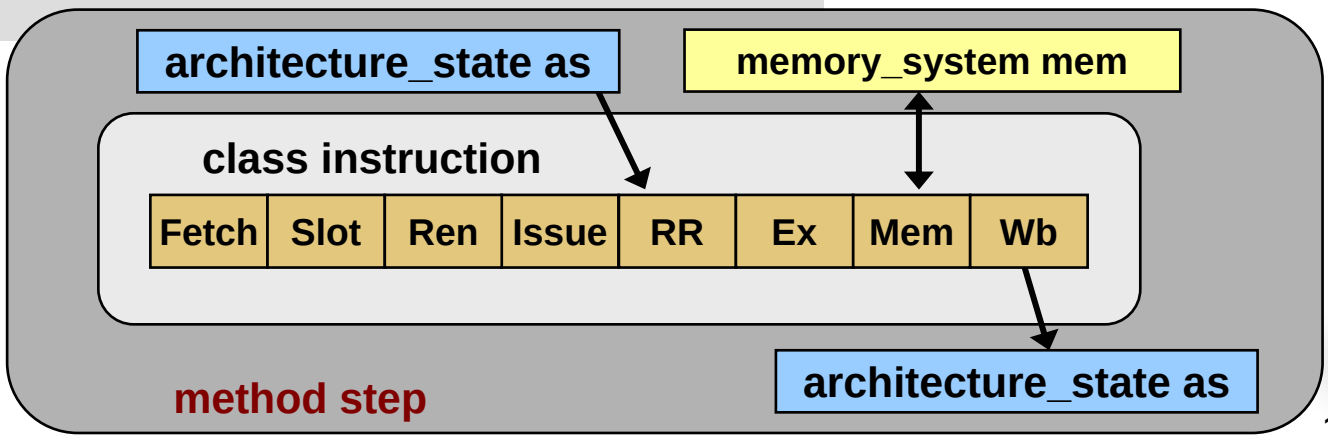
option program name

Class simple_chip method step

```
int simple_chip::step(){
  p->Fetch(&as->pc);     /* pipeline stage 0 */
  p->Slot();             /* pipeline stage 1 */
  p->Rename();           /* pipeline stage 2 */
  p->Issue();            /* pipeline stage 3 */
  p->RegisterRead();     /* pipeline stage 4 */
  p->Execute(&as->pc);   /* pipeline stage 5 */
  p->Memory();           /* pipeline stage 6 */
  p->WriteBack();

  /* split a conditional move,see README.txt */
  execute_cmovb(p, as);

  e->retired_inst++;
  house_keeper(sys, sc, e, deb);

  return sys->running;
}
```

One instruction is executed by calling **eight functions**.

They are corresponding to **seven pipeline stages** and then calling the eighth function of **WriteBack** in order



Class instruction definition

```
class instruction{
  evaluation_result  *e;
  architecture_state *as;
  system_manager     *sys;
  memory_system      *mem;

  INST_TYPE ir; /* 32bit instruction code  */
  int Op;       /* Opcode field            */
  int RA;       /* Ra field of the inst    */
  int RB;       /* Rb field of the inst    */
  int RC;       /* Rc field of the inst    */
  int ST;       /* store inst ?            */
  int LD;       /* load inst ?             */
  int LA;       /* load address inst ?     */
  int BR;       /* branch inst ?           */
  int Ai;       /* Rav is immediate ?      */
  int Bi;       /* Rbv is immediate ?      */
  int Af;       /* Rav from floating-reg ? */
  int Bf;       /* Rbv from floating-reg ? */
  int WF;       /* Write to the f-reg ?    */
  int WB;       /* Writeback reg index     */
  data_t  Npc;  /* Update PC or PC + 4     */
  data_t  Imm;  /* immediate               */
  data_t  Adr;  /* load & store address    */
  data_t  Rav;  /* Ra                      */
  data_t  Rbv;  /* Rb                      */
  data_t  Rcv;  /* Rc                      */
```

```
public:
  int Fetch(data_t *);
  int Fetch(data_t *, INST_TYPE);
  int Slot();
  int Rename();
  int Issue();
  int RegisterRead();
  int Execute(data_t *);
  int Memory();
  int WriteBack();

  INST_TYPE get_ir();
  int data_ld(data_t *, data_t *);
  int data_st(data_t *, data_t *);
  instruction(architecture_state *,
              memory_system *,
              system_manager *,
              system_config *,
              evaluation_result *);
};
```

Instruction fetch stage

```
int instruction::Fetch(data_t *pc){
  mem->ld_inst(pc, &ir);
  Npc.init(pc->ld() + 4);
  return 0;
}
```

- The code of an instruction fetch stage is shown.
- This function **loads 4 bytes instruction** from the memory specified by the program counter, and stores it in the **variable ir**.
- Then, the address of the next instruction is stored in variable Npc.

Instruction fetch stage sets some variables

```
class instruction{
  evaluation_result  *e;
  architecture_state *as;
  system_manager     *sys;
  memory_system      *mem;

  INST_TYPE ir; /* 32bit instruction code  */
  int Op;       /* Opcode field            */
  int RA;       /* Ra field of the inst    */
  int RB;       /* Rb field of the inst    */
  int RC;       /* Rc field of the inst    */
  int ST;       /* store inst ?            */
  int LD;       /* load inst ?             */
  int LA;       /* load address inst ?     */
  int BR;       /* branch inst ?           */
  int Ai;       /* Rav is immediate ?      */
  int Bi;       /* Rbv is immediate ?      */
  int Af;       /* Rav from floating-reg ? */
  int Bf;       /* Rbv from floating-reg ? */
  int WF;       /* Write to the f-reg ?    */
  int WB;       /* Writeback reg index     */
  data_t  Npc;  /* Update PC or PC + 4     */
  data_t  Imm;  /* immediate               */
  data_t  Adr;  /* load & store address    */
  data_t  Rav;  /* Ra                      */
  data_t  Rbv;  /* Rb                      */
  data_t  Rcv;  /* Rc                      */
```

```
public:
  int Fetch(data_t *);
  int Fetch(data_t *, INST_TYPE);
  int Slot();
  int Rename();
  int Issue();
  int RegisterRead();
  int Execute(data_t *);
  int Memory();
  int WriteBack();

  INST_TYPE get_ir();
  int data_ld(data_t *, data_t *);
  int data_st(data_t *, data_t *);
  instruction(architecture_state *,
              memory_system *,
              system_manager *,
              system_config *,
              evaluation_result *);
};
```

Slot (or decode) stage

```
int instruction::Slot(){
  Op  = (ir>>26) &  0x3F;
  RA  = (ir>>21) &  0x1F;
  RB  = (ir>>16) &  0x1F;
  RC  = (ir    ) &  0x1F;
  WF  = ((Op&MSK2)==0x14 || (Op&MSK2)==0x20);
  LA  = (Op==0x08 || Op==0x09);
  LD  = (Op==0x0a || Op==0x0b || Op==0x0c ||
         (Op&MSK2)==0x20 || (Op&MSK2)==0x28);
  ST  = (Op==0x0d || Op==0x0e || Op==0x0f ||
         (Op&MSK2)==0x24 || (Op&MSK2)==0x2c);
  BR  = ((Op&MSK4)==0x30);
  WB  = (LD || (Op&MSK2)==0x08 || Op==0x1a ||
         Op==0x30 || Op==0x34) ? RA :
        ((Op&MSK3)==0x10 || Op==0x1c) ? RC : 31;
  Af  = (Op==0x15 || Op==0x16 || Op==0x17 ||
         Op==0x1c ||
         (Op&MSK2)==0x24 || (Op&MSK3)==0x30);
  Bf  = ((Op&MSK2)==0x14);
  Ai = (Op==0x08 || Op==0x09 || LD);
  Bi = (BR || (Op&MSK2)==0x10 && (ir & BIT12));

  /** For the CMOV Split Code (CMOV1) **/
  if(cmov_ir_create(ir)){
    RB = RC;
    Bi = 0;
  }
  return 0;
}
```

- The code of a decode stage is shown.
- **The variables are decoded** using the fetched instruction code .
- The description of Verilog-HDL is similar to the code.
- Therefore, part of the C++ code can be reused for Verilog-HDL.

Slot (or decode) stage sets some variables

```
class instruction{
  evaluation_result  *e;
  architecture_state *as;
  system_manager     *sys;
  memory_system      *mem;

  INST_TYPE ir; /* 32bit instruction code  */
  int Op;       /* Opcode field            */
  int RA;       /* Ra field of the inst    */
  int RB;       /* Rb field of the inst    */
  int RC;       /* Rc field of the inst    */
  int ST;       /* store inst ?            */
  int LD;       /* load inst ?             */
  int LA;       /* load address inst ?     */
  int BR;       /* branch inst ?           */
  int Ai;       /* Rav is immediate ?      */
  int Bi;       /* Rbv is immediate ?      */
  int Af;       /* Rav from floating-reg ? */
  int Bf;       /* Rbv from floating-reg ? */
  int WF;       /* Write to the f-reg ?    */
  int WB;       /* Writeback reg index     */
  data_t  Npc;  /* Update PC or PC + 4     */
  data_t  Imm;  /* immediate               */
  data_t  Adr;  /* load & store address    */
  data_t  Rav;  /* Ra                      */
  data_t  Rbv;  /* Rb                      */
  data_t  Rcv;  /* Rc                      */
```

```
public:
  int Fetch(data_t *);
  int Fetch(data_t *, INST_TYPE);
  int Slot();
  int Rename();
  int Issue();
  int RegisterRead();
  int Execute(data_t *);
  int Memory();
  int WriteBack();

  INST_TYPE get_ir();
  int data_ld(data_t *, data_t *);
  int data_st(data_t *, data_t *);
  instruction(architecture_state *,
              memory_system *,
              system_manager *,
              system_config *,
              evaluation_result *);
};
```

Issue stage

```
int instruction::Issue(){
  DATA_TYPE Lit, D16, D21, tmp, d21e, d16e;
  d21e = ((ir & MASK21) | EXTND21) << 2;
  d16e  = (ir & MASK16) | EXTND16;

  Lit = (ir>>13) & 0xFF;
  D21 = (ir & BIT20) ? d21e : (ir&MASK21)<<2;
  D16 = (ir & BIT15) ? d16e : (ir&MASK16);
  if(Op==0x09) D16 = (D16 << 16);

  tmp = (LA||LD||ST) ? D16 : (BR) ? D21 : Lit;
  Imm.init(tmp);
  return 0;
}
```

- The code of an issue stage is shown.
- Here, **an immediate Imm** is generated according to the type of instruction.

Issue stage sets variable Imm

```
class instruction{
  evaluation_result  *e;
  architecture_state *as;
  system_manager     *sys;
  memory_system      *mem;

  INST_TYPE ir; /* 32bit instruction code  */
  int Op;       /* Opcode field            */
  int RA;       /* Ra field of the inst    */
  int RB;       /* Rb field of the inst    */
  int RC;       /* Rc field of the inst    */
  int ST;       /* store inst ?            */
  int LD;       /* load inst ?             */
  int LA;       /* load address inst ?     */
  int BR;       /* branch inst ?           */
  int Ai;       /* Rav is immediate ?      */
  int Bi;       /* Rbv is immediate ?      */
  int Af;       /* Rav from floating-reg ? */
  int Bf;       /* Rbv from floating-reg ? */
  int WF;       /* Write to the f-reg ?    */
  int WB;       /* Writeback reg index     */
  data_t  Npc;  /* Update PC or PC + 4     */
  data_t  Imm;  /* immediate               */
  data_t  Adr;  /* load & store address    */
  data_t  Rav;  /* Ra                      */
  data_t  Rbv;  /* Rb                      */
  data_t  Rcv;  /* Rc                      */
```

```
public:
  int Fetch(data_t *);
  int Fetch(data_t *, INST_TYPE);
  int Slot();
  int Rename();
  int Issue();
  int RegisterRead();
  int Execute(data_t *);
  int Memory();
  int WriteBack();

  INST_TYPE get_ir();
  int data_ld(data_t *, data_t *);
  int data_st(data_t *, data_t *);
  instruction(architecture_state *,
              memory_system *,
              system_manager *,
              system_config *,
              evaluation_result *);
};
```

Register read stage

```
int instruction::RegisterRead(){
  Rav = Ai ? Imm : Af ? as->f[RA] : as->r[RA];
  Rbv = Bi ? Imm : Bf ? as->f[RB] : as->r[RB];
  return 0;
}
```

- The code of a register read stage is shown.
- The **values of Rav and Rbv** are each selected from an immediate value, a floating point register and an integer register.

Register read stage sets some variables

```
class instruction{
  evaluation_result  *e;
  architecture_state *as;
  system_manager     *sys;
  memory_system      *mem;

  INST_TYPE ir; /* 32bit instruction code  */
  int Op;       /* Opcode field            */
  int RA;       /* Ra field of the inst    */
  int RB;       /* Rb field of the inst    */
  int RC;       /* Rc field of the inst    */
  int ST;       /* store inst ?            */
  int LD;       /* load inst ?             */
  int LA;       /* load address inst ?     */
  int BR;       /* branch inst ?           */
  int Ai;       /* Rav is immediate ?      */
  int Bi;       /* Rbv is immediate ?      */
  int Af;       /* Rav from floating-reg ? */
  int Bf;       /* Rbv from floating-reg ? */
  int WF;       /* Write to the f-reg ?    */
  int WB;       /* Writeback reg index     */
  data_t  Npc;  /* Update PC or PC + 4     */
  data_t  Imm;  /* immediate               */
  data_t  Adr;  /* load & store address    */
  data_t  Rav;  /* Ra                      */
  data_t  Rbv;  /* Rb                      */
  data_t  Rcv;  /* Rc                      */
```

```
public:
  int Fetch(data_t *);
  int Fetch(data_t *, INST_TYPE);
  int Slot();
  int Rename();
  int Issue();
  int RegisterRead();
  int Execute(data_t *);
  int Memory();
  int WriteBack();

  INST_TYPE get_ir();
  int data_ld(data_t *, data_t *);
  int data_st(data_t *, data_t *);
  instruction(architecture_state *,
              memory_system *,
              system_manager *,
              system_config *,
              evaluation_result *);
};
```

Execution stage

```
int instruction::Execute(data_t *Tpc){
  /*** Update Rcv ***/
  if(BR || Op==OP_JSR){
    Rcv=Npc;
  }
  else if(!LD){
    ALU(ir, &Rav, &Rbv, &Rcv);
  }

  /*** Update Adr ***/
  Adr.init(0);
  if(LD || ST){
    ALU(ir, &Imm, &Rbv, &Adr);
  }

  /*** Update Tpc ***/
  *Tpc = Npc;
  if(Op==OP_JSR){
    *Tpc = Rbv;
    Tpc->st(Tpc->ld() & ~3ull);
  }
  if(BR){
    BRU(ir, &Rav, &Rbv, &Npc, Tpc);
  }
  return 0;
}
```

- The code of an execution stage is shown.
- Three data values are updated in the execution stage.
- The arithmetic and logic instruction calculates the value of **Rcv** by considering **Rav and Rbv** as input.
- A load/store instruction calculates the memory reference **address Adr**.
- A branch instruction calculates the branch **target address Tpc**.

Memory access stage

```
int instruction::Memory(){
  if(ST) data_st(&Adr, &Rav);
  if(LD) data_ld(&Adr, &Rcv);
  return 0;
}
```

- The code of a memory access stage is shown.
- In the store instruction, the value of **Rav is stored** in memory.
- In the load instruction, the **loaded value is saved at Rcv**.

Execution and Memory access stages set some variables

```
class instruction{
  evaluation_result  *e;
  architecture_state *as;
  system_manager     *sys;
  memory_system      *mem;

  INST_TYPE ir; /* 32bit instruction code  */
  int Op;       /* Opcode field            */
  int RA;       /* Ra field of the inst    */
  int RB;       /* Rb field of the inst    */
  int RC;       /* Rc field of the inst    */
  int ST;       /* store inst ?            */
  int LD;       /* load inst ?             */
  int LA;       /* load address inst ?     */
  int BR;       /* branch inst ?           */
  int Ai;       /* Rav is immediate ?      */
  int Bi;       /* Rbv is immediate ?      */
  int Af;       /* Rav from floating-reg ? */
  int Bf;       /* Rbv from floating-reg ? */
  int WF;       /* Write to the f-reg ?    */
  int WB;       /* Writeback reg index     */
  data_t  Npc;  /* Update PC or PC + 4     */
  data_t  Imm;  /* immediate               */
  data_t  Adr;  /* load & store address    */
  data_t  Rav;  /* Ra                      */
  data_t  Rbv;  /* Rb                      */
  data_t  Rcv;  /* Rc                      */
```

```
public:
  int Fetch(data_t *);
  int Fetch(data_t *, INST_TYPE);
  int Slot();
  int Rename();
  int Issue();
  int RegisterRead();
  int Execute(data_t *);
  int Memory();
  int WriteBack();

  INST_TYPE get_ir();
  int data_ld(data_t *, data_t *);
  int data_st(data_t *, data_t *);
  instruction(architecture_state *,
              memory_system *,
              system_manager *,
              system_config *,
              evaluation_result *);
};
```

Writeback stage

```
int instruction::WriteBack(){
  if(Op==OP_PAL){
    sys->execute_pal(this);
  }

  if(!WF && WB!=31) as->r[WB] = Rcv;
  if( WF && WB!=31) as->f[WB] = Rcv;
  return 0;
}
```

- The code of a writeback stage is shown.
- In the instruction which generates a result, **Rcv is stored in a register file.**
- **The instruction completes execution.**

- An execute_pal function is called when the instruction currently executed is PAL(Privileged Architecture Library) code.

SimCore/Alpha Version 1.0 design policy again

- Target applications
  - SPEC CINT95 and CINT2000.
- **Simple and readable (enjoyable and easy to read) code**
  - **No global variable**
  - **No goto statement**
  - **No conditional compilation**
  - **The source code is only about 2,800 lines in C++.**
- It offers another choice
  - A processor simulator is an important tool.
  - It is advantageous to choose a suitable tool from many choices.
  - SimCore/Alpha offers another choice.
- Functional simulator, but
  - Although only the capability of a functional simulator is given, the code is written considering the extension to the out-of-order processor simulator.

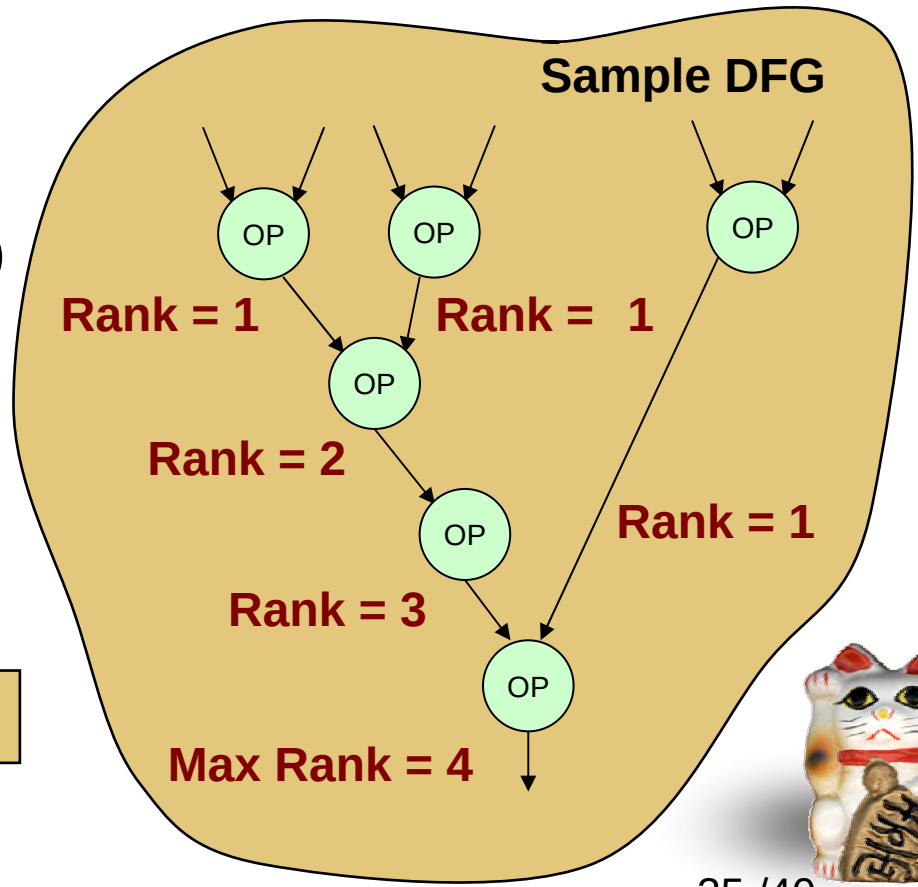
SimCore/Alpha practical use

This section gives an example of the SimCore/Alpha practical use.

SimCore/Alpha is modified to measure ideal instruction-level parallelism. The parallelism is acquired only after considering data dependency as a restriction.

To measure ideal instruction-level parallelism

- Ideal instruction-level parallelism or **Oracle IPC** is the parallelism which is acquired only after considering data dependency as a restriction.
  - **No control dependency**
  - **No resource conflict**
- In order to measure Oracle IPC, the value (this is called the **rank**) equivalent to the height of the data flow graph is calculated.



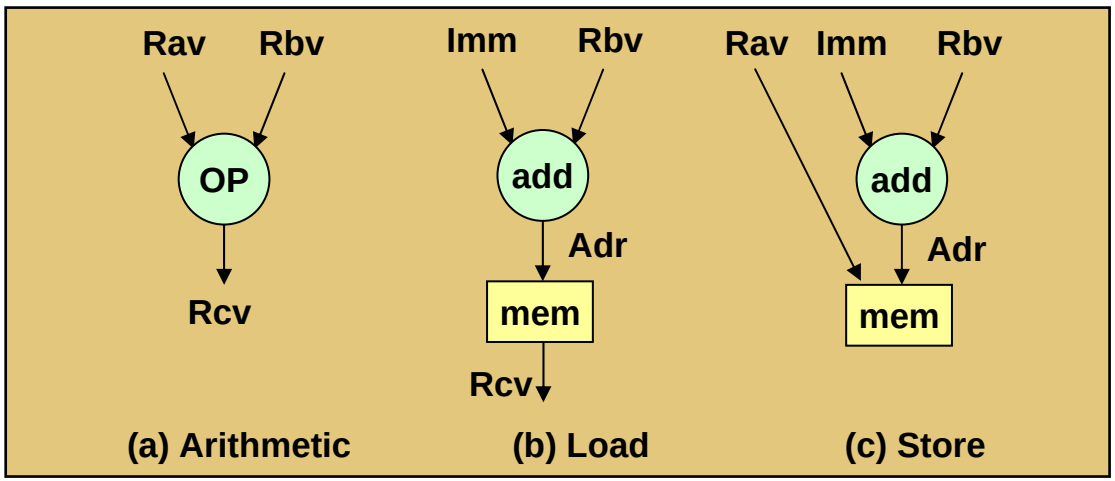
Oracle IPC = 6/4 = 1.5

Calculation of the rank for each instruction type

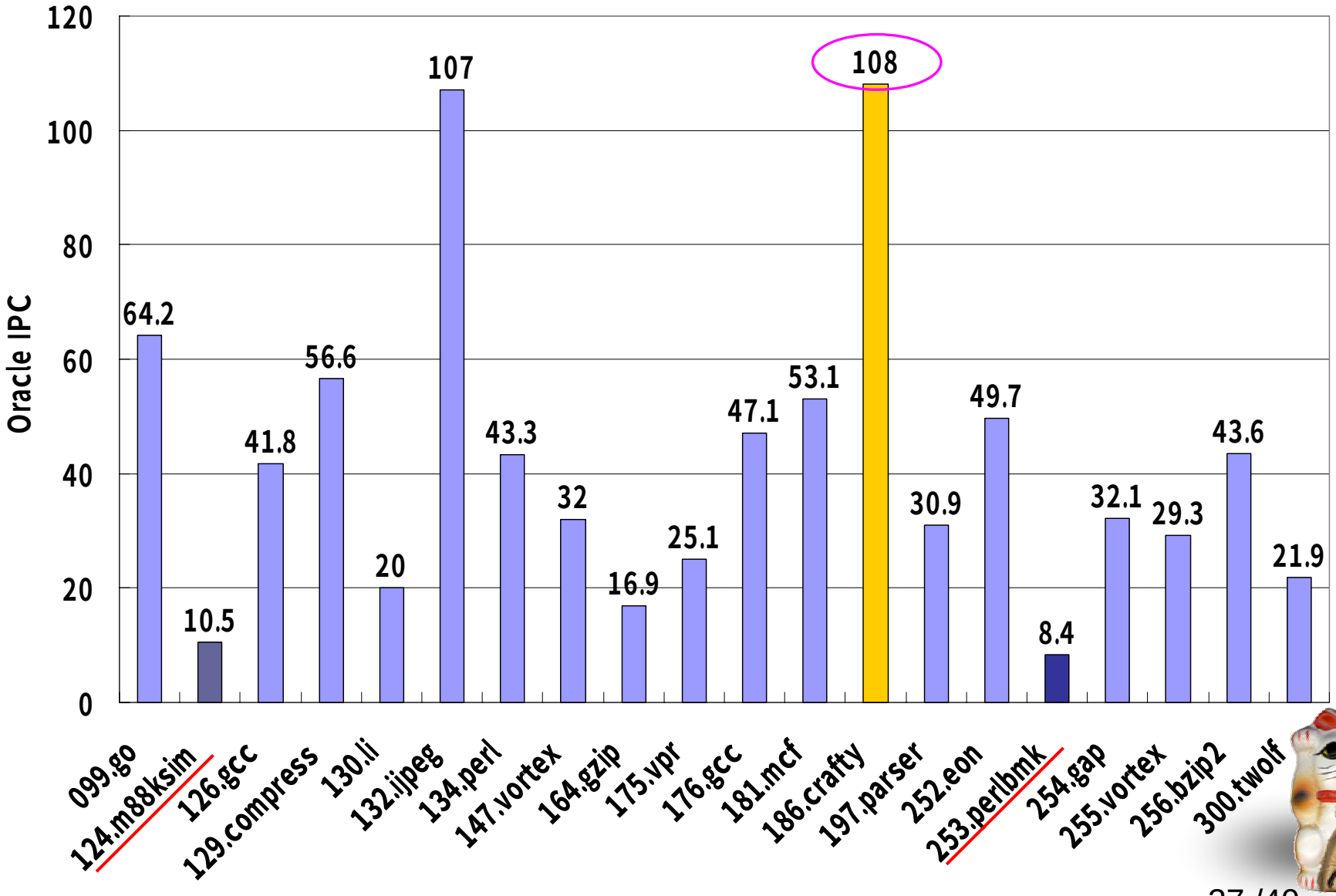
- Class data_t is modified to store **the rank.**
- **Arithmetic and logic instruction**, the rank of Rcv is obtained by adding the operation latency to the maximum of the rank of Rav and Rbv.
- **Load instruction**, the rank is calculated by adding the memory reference latency and the address computation latency to the rank of Rbv.
- **Store instruction**, the maximum of the Rav written in memory and the rank obtained by address computation is the rank of the data.

```
class data_t{
  uint64_t value;
 public:
  int cmov;
  uint32_t rank;
  uint64_t ld();
  int   st(uint64_t);
  int init(uint64_t);
};
```

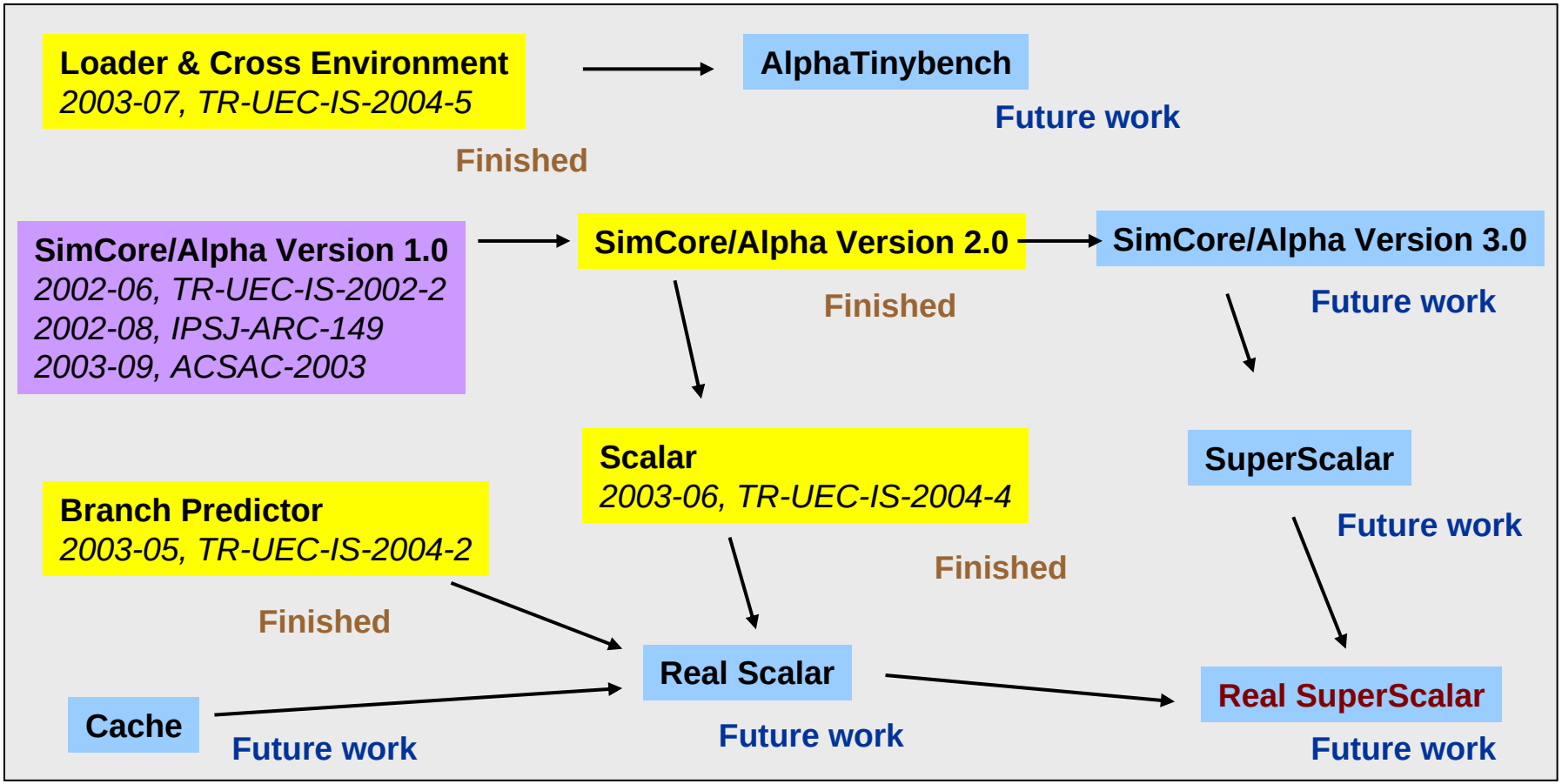
Modified class data_t



Oracle IPC measured by the modified SimCore/Alpha



SimCore/Alpha development plan



SimCore/Alpha Version 1.0 is a functional simulator.
We have the plan to construct cycle-accurate performance simulator **SimCore/Alpha Real SuperScalar** modeling various out-of-order superscalar processors.

Summary

- Processor simulator **SimCore/Alpha Version 1.0** was developed for processor architecture research and processor education.
- To show the high **readability** of the code, the software architecture of SimCore/Alpha was explained using the actual C++ code.
- As an example of the practical use of SimCore/Alpha, the evaluation method of **Oracle-IPC** was explained.

- We have the plan to construct **cycle-accurate performance simulators** modeling various out-of-order superscalar processors.
- We are implementing SimCore/Alpha of the **Verilog-HDL version**, which works on an FPGA board.

SimCore/Alpha is available!

- **SimCore Homepage**
  - The source code of SimCore/Alpha is **downloadable** from the following URL.
    http://www.yuba.is.uec.ac.jp/~kis/SimCore/
- **SimCore/Alpha document**
  - **SimAlpha: Alpha Processor Simulator with Readable Source Code**, Technical Report UEC-IS-2002-2
  - **Implementation of Typical Branch Predictors for High Performance Microprocessors**, Technical Report UEC-IS-2003-2
  - **Implementation and Verification of Scalar Processor Simulator**, Technical Report UEC-IS-2003-4
  - **Implementation of SimAlpha-Loader and Construction of Cross-Development Environment**, Technical Report UEC-IS-2003-5

Note that SimCore/Alpha was called SimAlpha until 2003-09.