*Technical Report UEC-IS-2003-11, Version 2003-12-10*

# Implementation of a Simple and Readable Processor Simulator

Kenji Kise, Takahiro Katagiri, Hiroki Honda, and Toshitsugu Yuba
Graduate School of Information Systems,
The University of Electro-Communications

We have developed a processor simulator SimAlpha Version 1.0 for research and education activities. Its design policy is to keep the source code readable (*enjoyable and easy to read* ) and simple. SimAlpha is written in C++ and the source code consists of only 2,800 lines. This paper describes the software architecture of SimAlpha by referring to its source code. To show an example of SimAlpha in practical use, we present the ideal instruction-level parallelism of SPEC CINT95 and CINT2000 benchmarks measured with a modified version of SimAlpha.

**Key-words** processor simulator, SimAlpha, simple and readable

## 1 Introduction

Various processor simulators[2, 7] are used as tools for processor architecture research or processor education. The environment in which a processor simulator can perform is improving dramatically due to the increased speed of PCs and the growing use of PC clusters. However, the time needed for simulator construction increases as the architectural idea to be implemented increases in complexity. In many cases the evaluation finishes within several weeks, although several months are needed for the construction of the simulator, even if the simulator is developed with existing tools. SimpleScalar Tool Set[4] is a famous processor simulator used for purposes such as processor research and education. But, since SimpleScalar can be implemented in high-speed simulations, it is not a code that can easily be modified.

SimAlpha Version 1.0 is an Alpha[6] processor simulator. Its code is easy to understand and easy to modify. SimAlpha has a function equivalent to the functional simulator of SimpleScalar/Alpha or a sim-safe program. Although it is not the clock-level simulator of pipeline processing or out-of-order execution, the described code should be considered an extension to these.

SimAlpha has a different policy from SimpleScalar. The SimAlpha simulator is described from scratch. It uses C++ and the code size is small at about 2,800 lines. In order to make it readable, neither global variables, goto statements nor conditional compilation is used. The aim of SimAlpha is to show the implementation of a processor simulator with a different policy. A processor simulator is an important tool, and it is advantageous to choose the most suitable tool, given many choices. As a tool for processor research and education, SimAlpha offers another choice.

## 2 Preparation of SimAlpha

This section explains the structure of an original execution image file, the simulation speed, and the verification policy of SimAlpha.

### 2.1 Execution image file

To run SimAlpha, application or benchmark programs have to be prepared. SimAlpha reads an execution image file in its original format. It does not read Alpha binary files. By adapting the simple original format, knowledge of executable formats such as ELF and COFF is not necessary.

```
/* SimAlpha 1.0 Image File */
/*** Registers ***/
/@reg 16 0000000000000003
/@reg 17 000000011ff97008
/@pc  32 0000000120007d80
/*** Memory    ***/
@11ff97008 11ff97188
```

Figure 1: Sample SimAlpha execution image file.

An example of an execution image file is shown in Figure 1. This execution image file is in text format and consists of two parts. It is created from an Alpha binary file. In the first part, values are assigned to some of the registers. In the example of Figure.1, the hexadecimal value 3 is assigned to the 16th register, the value 11ff97008 is assigned to the 17th register, and the value of 120007d80 is assigned to a program counter. Registers without these specifications are initialized with the value 0. Moreover, all of the floating point registers are also initialized with the value 0. In the second part, the value of some memory is assigned in the same man-

ner. In the example of Figure.1, the value 11ff97188 is assigned to the memory of address 11ff97008. The content of all unspecified memory is initialized with the value 0.

### 2.2 Benchmark programs and organization of PC used for evaluations

A total of 20 benchmark programs, including 8 from SPEC CINT95 and 12 from CINT2000[1], are used to evaluate SimAlpha for this paper. The reduced input set of MinneSPEC[3] from the University of Minnesota is used on the 9 benchmarks of CINT2000. In the other benchmarks, an input parameter is adjusted so that the number of simulated instructions is reduced. The binary of SPEC CINT95 is generated using a DEC C compiler with the optimization option of O4. The binaries of SPEC CINT2000 are downloaded from the SimpleScalar web site.

Table 1: Organization of PC used for evaluations.

| | |
|---|---|
| Machine | GS-SR101 1U Rackmount Server |
| Processor | Intel Pentium III 1GHz × 2 |
| DRAM | 512MB (PC133/3/R-ECC × 1) |
| HDD | Seagate ST380021A × 2 |
| OS | Red Hat Linux 7.2 |

Data such as simulation speed is measured using the Pentium III 1GHz PC. The organization of the PC is summarized in Table 1. The executed instructions of each benchmark are summarized in the second column of Table.2.

### 2.3 Simulation speed of SimAlpha

The compiler of egcs-1.1.2 with the optimization option of O2 is used to compile SimAlpha.

SimAlpha has a function equivalent to the functional simulator of SimpleScalar/Alpha or a sim-safe program. We ran the 20 benchmark programs on SimAlpha and sim-safe, and calculated the average simulation speed. The simulation speed for SimAlpha is 1.1 MIPS (Million Instructions Per Second), compared to 3.1 MIPS for sim-safe.

It is a drawback of SimAlpha that a simulation takes about 3 times as long as a SimpleScalar simulation. However, in many cases the development of a simulator dominates project time. If the time of simulator development can be shortened, the slow simulation speed does not become a problem.

### 2.4 Verification of SimAlpha

During the development of SimAlpha, compatibility with SimpleScalar was carefully confirmed.

Whenever the simulator executed one instruction, all values of the architecture state (a program counter, 32 integer registers, 32 floating point registers) of SimAlpha and the architecture state of SimpleScalar were compared. We confirmed that the two architecture states were identical during the 20 benchmark simulations.

In order to simplify the verification procedure, a way to embed the object of SimAlpha into another simulator is offered. Moreover, since SimAlpha does not use any global variables, two or more simulation images can easily be generated in one process. By using these functions, any bug of the simulator under development is discovered at an early stage. Also, by using these functions one can confirm the justification of the simulator.

## 3 SimAlpha internals

In this section, in order to show the high readability of the source code, the internal structure of SimAlpha is explained showing actual C++ code (not pseudocode).

First, we start with an explanation of the main function. Then, we explain how the constructor of the object chip generates seven objects. After seeing the definition of some important classes, the definition and code of the class instruction, which play an important role, are explained.

### 3.1 Main function

The main function of SimAlpha is shown.

```
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;
}
```

After setting the program name and options, the chip of a simple_chip type object is generated. The member function step, which executes one instruction and returns the value of 0 when all of the instructions to be executed have been consumed (when the simulation has been completed). The simulation is advanced by repeating the while loop until the function step returns

the value 0. When the loop finishes, the object chip is released, and its destructor displays the simulation result.

## 3.2 Class simple_chip

The definition and constructor of class simple_chip are shown.

```
class simple_chip{
  system_config      *sc;
  evaluation_result  *e;
  debug              *deb;
  system_manager     *sys;
  instruction        *p;
 public:
  memory_system      *mem;
  architecture_state *as;
  simple_chip(char *, char **);
  ~simple_chip();
  int step();
};

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);
}
```

The constructor of a simple_chip generates seven objects. The destructor displays the simulation result, and then it releases the seven objects.

The code of the member function step of class simple_chip, which performs the stepwise execution, is shown.

```
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 seven functions corresponding to seven pipeline stages and then calling the eighth function of WriteBack in order. Although only the capability of a function-level simulator is offered in SimAlpha Version 1.0, in consideration of the readability and extendibility of a code, the operation of an instruction was divided and described for eight stages, referring to the instruction pipeline of Alpha21264[6].

A conditional move instruction (CMOV instruction) is split into two new instructions for two input operands. Function execute_cmovb processes the second split instruction of the CMOV instruction.

## 3.3 Definition of some important classes

### 3.3.1 Class data_t expressing data

The calculation results are stored in a register file or memory. These results are defined as the collection of class data_t objects. The definition and code of class data_t are shown.

```
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;
}
```

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.

### 3.3.2 Architecture state

The definition and constructor of the class architecture_state, which consists of a program counter, an integer register, and floating point registers, are shown.

```
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 *);
};
```

### 3.3.3 Class evaluation_result

The data under evaluation is saved in an evaluation_result type object. Although the value of the evaluation_result type object is updated during the simulation, these values do not affect the behavior of the simulation. The definition of class evaluation_result is shown.

```
class evaluation_result{
 public:
  uint64_t retired_inst;
  int used_memory_block;
  time_t time_begin;     /* start time stamp */
  struct timeval tp;     /* start time stamp */
  struct timezone tzp;   /* start time stamp */
  evaluation_result();
};
```

Each variable stores the executed number of instructions, the number of pages used in the main memory, and the time when the simulation started.

### 3.3.4 Class system_config

Information on system configuration is stored in a system_config type object. These values are defined before the start of the simulation and, in principle, do not change during the simulation.

## 3.4 Class instruction

This section explains the definition and code of the class instruction. Since the function Rename has no code, its explanation is omitted. The definition of the class instruction is shown.

```
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 *);
};
```

The values of the private variables are calculated as the function corresponding to the pipeline stages are called, and the processing of the instruction progresses. Fourteen variables defined as the int type hold the decoded value from the instruction code ir. A data_t type variable holds the value loaded from the memory or registers files, or holds the value to be stored in the memory or register files.

### 3.4.1 Instruction fetch stage

The code of an instruction fetch is shown.

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

```
int instruction::Fetch(data_t *pc, INST_TYPE ir_t){
  ir = ir_t;
  Npc.init(pc->ld());
  return 0;
}
```

Two Fetch functions exist. The code shown above is the function Fetch for the usual instruction (instruction other than CMOV). This function loads 4 bytes of instruction from the address which the program counter specifies, and stores it in the variable ir. Then, the address of the next instruction is stored in Npc.

The code shown below is used to fetch the second split instruction in a conditional move instruction. Therefore, the function Fetch will be called with the instruction code as one of the arguments.

### 3.4.2 Slot stage

The code of a slot stage is shown.

```
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 values of some variables are decoded using the instruction code fetched in the previous stage. Instead of assignment of the decoded values to variables, the code can be described using a macro. Although an improvement in simulation time is expected by using a macro, the method of variable assignment was chosen for code readability. The description of Verilog-HDL is similar to the above description. Therefore, part of the C++ code can be reused for Verilog-HDL.

### 3.4.3 Issue stage

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

```
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;
}
```

### 3.4.4 Register read stage

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 file, and an integer register file.

```
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;
}
```

### 3.4.5 Execution stage

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.

```
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;
}
```

### 3.4.6 Memory access stage

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.

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

### 3.4.7 Writeback stage

The code of a writeback stage is shown. In the instruction which generates a result, Rcv is stored in a register file, and the instruction completes execution. An execute_pal function is called when the instruction currently executed is PAL(Privileged Architecture Library) code.

```
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;
}
```

## 3.5 Memory system

The memory system of SimAlpha Version 1.0 does not contain cache. It is implemented as a simple organization of the main memory only. The address of the Alpha AXP architecture is 64 bits in width. But, in SimAlpha Version 1.0, 32 bits of the higher ranks of an address are disregarded, and only 32 bits of the low rank are used. In the code generated by the compiler, since the value of the higher 32 bits is fixed to 0x00000001, it does not become a problem by such implementation.

The definition of class memory_system is shown.

```
class memory_system{
  evaluation_result *e;
  class main_memory *mm;
  void ld_8byte(data_t *, data_t *);
  void st_8byte(data_t *, data_t *, DATA_TYPE);
 public:
  void ld_inst(data_t *, INST_TYPE *);
  void ld_nbyte(int, data_t *, data_t *);
  void st_nbyte(int, data_t *, data_t *);
  ~memory_system();
  memory_system(system_config *,
                evaluation_result *);
};
```

### 3.5.1 Implementation of main memory

The main memory is implemented as an array of the data_t type object. 32-bit memory space is expressed as a collection of 8-KB pages.

The definition of main memory is shown.
Array block_table, which stores the page pointer, holds the number of entries that divide the 32-bit address by the page size of 8 KB. In the constructor of the main memory, all the entries of array block_table are initialized by NULL. The main memory is referred to using function ld_8byte to load 8 bytes, and function st_8byte, which implements the store with a mask.

```
class main_memory{
  evaluation_result *e;
  data_t *block_table[BLOCK_TABLE_SIZE];
  data_t *allocblock(data_t *);
 public:
  void ld_8byte(data_t *, data_t *);
  void st_8byte(data_t *, data_t *, DATA_TYPE);
  main_memory(evaluation_result *);
};
```

The code of the main memory is shown. In function ld_8byte, the address to load and the pointer which stores the loaded data are specified as arguments.

Function allocblock is called when the page containing the specified address is the first reference. Data is loaded after function allocblock assigns the page.

In function ld_8byte, memory address, the pointer of data to be stored and the mask are specified as arguments.

```
main_memory::main_memory(evaluation_result *ev){
  e = ev;
  for(int i=0; i<BLOCK_TABLE_SIZE; i++)
    block_table[i]=NULL;
```

```
}

data_t *main_memory::allocblock(data_t *a){
  data_t *ret=
    (data_t *)calloc(BLOCK_SIZE/DATA_T_SIZE,
                     sizeof(class data_t));
  block_table[MM_TABLE_INDEX(a->ld())]=ret;
  if(ret==NULL){
    printf("** Error in allocblock.\n");
    exit(0);
  }
  e->used_memory_block++;
  return ret;
}

void main_memory::ld_8byte(data_t *a, data_t *d){
  ADDR_TYPE adr = a->ld() & ~7;
  data_t *ptr = block_table[MM_TABLE_INDEX(adr)];
  unsigned int offset =
    (adr & BLOCK_MASK)/DATA_T_SIZE;
  if(ptr==NULL) ptr=allocblock(a);
  *d = *(ptr + offset); /** COPY **/
}

void main_memory::st_8byte(data_t *a, data_t *d,
                           DATA_TYPE msk){
  ADDR_TYPE adr = a->ld() & ~7;
  data_t *ptr = block_table[MM_TABLE_INDEX(adr)];
  unsigned int offset =
    (adr & BLOCK_MASK)/DATA_T_SIZE;
  if(ptr==NULL) ptr=allocblock(a);
  (ptr + offset)->st(((ptr + offset)->ld() & msk)
                     | d->ld());
}
```

### 3.5.2 Implementation of memory system

The main memory shown previously allows only an 8-byte reference. Reference to the memory, including 1-byte, 2-byte and 4-byte units, is implemented as a function of the memory system.

The code of ld_nbyte is shown. The number of bytes to be loaded is specified in the first argument.

```
void memory_system::ld_nbyte(int n,
                             data_t *a, data_t *d){
  if(a->ld()%n!=0)
    printf("*** ld_nbyte n=%d miss-alignment.\n",
            n);

  ld_8byte(a, d);

  int offset = a->ld() & 7;
  switch(n){
  case 1: {
    DATA_TYPE data= (d->ld() >> (offset * 8))
                    & 0xfflu;
    d->st(data);
    break;
  }
  case 2: {
    DATA_TYPE data= (d->ld() >> (offset * 8))
                    & 0xffffllu;
    d->st(data);
    break;
  }
  case 4: {
    DATA_TYPE data= (d->ld() >> (offset * 8))
                    & 0xffffffffllu;
    d->st(data);
    break;
  }
  case 8: {
    break;
  }
  default:
    printf("Case %d, Error in load_nbyte\n", n);
    exit(1);
  }
}
```

The code of st_nbyte is shown. The number of bytes to be stored is specified in the first argument.

```
void memory_system::st_nbyte(int n, data_t *a,
                             data_t *d){
  if(a->ld()%n!=0)
    printf("*** st_nbyte n=%d miss-alignment.\n",
            n);

  int offset = a->ld() & 7;
  DATA_TYPE mask = 0;

  switch(n){
  case 1: {
    mask = ~(0xfflu << offset*8);
    DATA_TYPE data = (d->ld() & 0xfflu)
                     << offset*8;
    d->st(data);
    break;
  }
  case 2: {
    mask = ~(0xffffllu << offset*8);
    DATA_TYPE data = (d->ld() & 0xffffllu)
                     << offset*8;
    d->st(data);
    break;
  }
  case 4: {
    mask = ~(0xffffffffllu << offset*8);
    DATA_TYPE data = (d->ld() & 0xffffffffllu)
                     << offset*8;
    d->st(data);
    break;
  }
  case 8: {
```

```
    mask = 0;
    break;
  }
  default:
    printf("Case %d, Error in store_nbyte\n",
           n);
    exit(1);
  }

  st_8byte(a, d, mask);
}
```

# 4 Practical use of SimAlpha

This section gives an example of the SimAlpha practical use. SimAlpha is modified to measure ideal instruction-level parallelism. The parallelism is acquired only after considering data dependency as a restriction. The value to be measured has the same meaning as the Oracle instruction per cycle in [5].

## 4.1 Extension of class data_t

The data treated by SimAlpha is defined as a data_t type object, not as a standard unsigned long long type value. In order to measure ideal instruction-level parallelism, class data_t is modified so that the value (this will be called the rank) equivalent to the height of the data flow graph is calculated and stored.

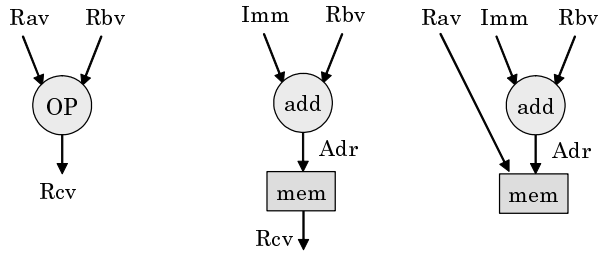
Physical memory is defined as an array of the object of class data_t. Since a load-and-store instruction refers to memory with a granularity of 1-8 bytes, there are some choices in the granularity that expresses the rank of the data in memory. Here, data with the 8-byte aligned unit is defined as one object.

The definition of class data_t, modified to measure ideal instruction-level parallelism, are shown. The uint32_t type variable rank was added to class data_t. The rank is stored in this variable. In the constructor, the variable rank is initialized by the value 0.

```
class data_t{
  uint64_t value;
 public:
  int cmov;
  uint32_t rank;  /* This line is inserted. */
  uint64_t ld();
  int   st(uint64_t);
  int init(uint64_t);
};
```

## 4.2 Calculation method of rank and ideal instruction level parallelism

The calculation method of a rank is shown in Figure 2. When an arithmetic and logic instruction is executed, the rank of output data Rcv is obtained by adding the operation latency to the maximum of the rank of the two input operands, Rav and Rbv. In the load instruction, rank is calculated by adding the memory reference latency and the address computation latency to the rank of Rbv. In the store instruction, the maximum of the Rav data written in memory and the rank obtained by address computation is considered to be the rank of the data.




(a) Arithmetic, Logic (b) Load (c) Store

(a) rank(Rcv) = max(rank(Rav), rank(Rbv)) + latency(OP)
(b) rank(Rcv) = rank(Rbv) + latency(add) + latency(mem)
(c) rank = max(rank(Rav), rank(Rbv) + latency(add))

Figure 2: The calculation method of the rank for each instruction type.

During a simulation, the maximum rank of all the data is updated apart from the rank for each of the data. The maximum of the ranks at the time when a simulation is completed expresses the height of the data flow graph, whose nodes are all the executed instructions. Therefore, ideal parallelism can be calculated from the number of executed instructions and the height of the data flow graph (the maximum of the ranks). The restriction that the data cannot be moved across a system call is added.

In the following evaluations, operation latency and memory reference latency are assumed to be one clock cycle when calculating a rank.

## 4.3 Extension of SimAlpha

SimAlpha was modified in order to measure ideal parallelism. Many portions of the modification consist of the calculation of a rank at the time the data is being generated. Only 26 lines of code is modified.

Except for the function st_8byte and the code which displays the result, the modified code is explained. The

comment /* Added */ in the code indicates that the line has been appended.

The code of the modified execute stage is shown. After the calculation in ALU, the addition of the operation latency 1 to the maximum of the rank of the Rav and Rbv is assigned as a rank of the Rcv.

```
int instruction::Execute(data_t *Tpc){
  /*** Update Rcv ***/
  if(BR || Op==OP_JSR){
    Rcv=Npc;
  }
  else if(!LD){
    ALU(ir, &Rav, &Rbv, &Rcv);
    Rcv.rank = (Rav.rank>Rbv.rank) ?  /* Added */
               Rav.rank : Rbv.rank;  /* Added */
    Rcv.rank += 1; /* ALU latency */  /* Added */
  }

  /*** Update Adr ***/
  Adr.init(0);
  if(LD || ST){
    ALU(ir, &Imm, &Rbv, &Adr);
    Adr.rank = (Imm.rank>Rbv.rank) ?  /* Added */
               Imm.rank : Rbv.rank;  /* Added */
    Adr.rank += 1; /* ALU latency */  /* Added */
  }

  /*** 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 the modified memory stage is shown. In the store instruction, the rank of the data is calculated before storing Rav. In the load instruction, the code which calculates the rank of the loaded Rcv data is appended.

```
int instruction::Memory(){
  if(ST){
    Rav.rank = (Adr.rank > Rav.rank) ? /* Added */
               Adr.rank : Rav.rank;   /* Added */
    if(Rav.rank <e->systemcall_rank)   /* Added */
        Rav.rank = e->systemcall_rank; /* Added */
    data_st(&Adr, &Rav);
  }
  if(LD){
    data_ld(&Adr, &Rcv);
    Rcv.rank = (Adr.rank>Rcv.rank) ?   /* Added */
               Adr.rank : Rcv.rank;    /* Added */
    Rcv.rank += 1; /* Load latency */  /* Added */
  }
  return 0;
}
```

The code of the modified writeback stage is shown. If data is copied to a register file, the maximum of the ranks is calculated. Moreover, since instruction scheduling over a system call is forbidden, the rank of the data cannot become smaller than the maximum rank at the time of the last system call.

```
int instruction::WriteBack(){
  if(Op==OP_PAL){
    sys->execute_pal(this);
    e->systemcall_rank = e->max_rank;  /* Added */
  }

  if(WB!=31){                          /* Added */
    if(e->max_rank < Rcv.rank)         /* Added */
       e->max_rank = Rcv.rank;         /* Added */
    if(Rcv.rank < e->systemcall_rank)  /* Added */
       Rcv.rank = e->systemcall_rank;  /* Added */
  }
  if(!WF && WB!=31) as->r[WB] = Rcv;
  if( WF && WB!=31) as->f[WB] = Rcv;
  return 0;
}
```

## 4.4 Evaluation result of ideal instruction level parallelism

The measurement result of ideal instruction-level parallelism (ILP) is shown in Table 2. We also summarize the executed code and the simulation speed (MIPS) in Table 2.

The amount of accessed memory during the simulation increases by appending the variable rank, as shown in the modified class data_t. Moreover, in spite of the increased processing for calculating a rank, a serious increase was not seen at simulation time. The simulation speed after modification was about 1.1 MIPS.

The measurement results of ideal instruction-level parallelism showed low parallelism in 124.m88ksim and 253.perlbmk. In the other benchmark, parallelism exceeding 15 was shown and we confirmed the high parallelism of 108 in 186.crafty. The data shown here is important in order to know the potential parallelism of a program. In addition, it can also be used for preliminary evaluations of the compilers or of compiler optimizations.

Table 2: The number of executed instructions, simulation speed, ideal instruction level parallelism measured using the modified SimAlpha.

| Program | instruction count | MIPS | ILP |
|---|---|---|---|
| 099.go | 138 million | 1.12 | 64.2 |
| 124.m88ksim | 127 million | 1.12 | 10.5 |
| 126.gcc | 150 million | 1.12 | 41.8 |
| 129.compress | 142 million | 1.14 | 56.6 |
| 130.li | 208 million | 1.11 | 20.0 |
| 132.ijpeg | 172 million | 1.21 | 107.0 |
| 134.perl | 153 million | 1.10 | 43.3 |
| 147.vortex | 184 million | 1.07 | 32.0 |
| 164.gzip | 596 million | 1.19 | 16.9 |
| 175.vpr | 17 million | 1.00 | 25.1 |
| 176.gcc | 551 million | 1.10 | 47.1 |
| 181.mcf | 188 million | 1.12 | 53.1 |
| 186.crafty | 4,264 million | 1.10 | 108.0 |
| 197.parser | 611 million | 1.10 | 30.9 |
| 252.eon | 94 million | 0.93 | 49.7 |
| 253.perlbmk | 200 million | 1.05 | 8.4 |
| 254.gap | 1,169 million | 1.12 | 32.1 |
| 255.vortex | 147 million | 1.06 | 29.3 |
| 256.bzip2 | 1,819 million | 1.12 | 43.6 |
| 300.twolf | 91 million | 1.00 | 21.9 |

In the example, class data_t is modified to store a rank. By extending SimAlpha using the same technique, the memory and branch behavior can be obtained.

# 5 Summary

The processor simulator SimAlpha Version 1.0 was developed for processor architecture research and processor education. In this paper, in order to show the high readability of the code, the software architecture of SimAlpha was explained using the actual C++ code.

As an example of the practical use of SimAlpha, the evaluation method of ideal instruction-level parallelism was explained. The function for measuring ideal instruction-level parallelism was implemented with a small code modification of only 26 lines. The ideal IPC of SPEC CINT95 and CINT2000 was measured using the modified version of SimAlpha, and the result was reported.

Historically, the development of SimAlpha for the C version began in March, 1999. Development of SimAlpha for the C++ version began in June, 1999. Now we are implementing SimAlpha of the Verilog-HDL version, which works on an FPGA board. This version will be helpful when simulation speed is important.

SimAlpha Version 1.0 is a function level simulator. We have the plan to construct cycle-accurate performance simulators modeling various out-of-order superscalar processors. It is another challenge to implement the complex processor models with readable and simple source code.

The source code of SimAlpha Version 1.0 and the source code of the modified version of SimAlpha to evaluate ideal instruction-level parallelism are downloadable from the following URL.

`http://www.yuba.is.uec.ac.jp/~kis/SimAlpha/`

# References


[1] Standard Performance Evaluation Corporation. SPEC benchmark suites. http://www.spec.org/.

[2] The MicroLib.org Project Homepage. http://www.microlib.org/.

[3] AJ KleinOsowski and David J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. In *Computer Architecture Letters*, volume 1, June 2002.

[4] Doug Burger and Todd M. Austin. The Simplescalar Tool Set, Version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin-Madison, June 1997.

[5] Monica S. Lam and Robert P. Wilson. Limits of Control Flow on Parallelism. In *19th Annual International Symposium on Computer Architecture*, pages 46–57, May 1992.

[6] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):25–36, March 1999.

[7] Shubhendu S. Mukherjee, Sarita V. Adve, Todd Austin, Joel Emer, and Peter S. Magnusson. Performance Simulation Tools. *IEEE Computer*, 35(2):38–39, February 2002.