Using the DEVS Paradigm to Implement a Simulated Processor

Sergio Daicz AlejandroTróccoli Sergio Zlotnik Gabriel Wainer

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Pabellón I - Ciudad Universitaria
Buenos Aires (1428) – ARGENTINA
gabrielw@dc.uba.ar
http://www.dc.uba.ar/people/proyinv/celldevs

Abstract

This work is devoted to present the design and implementation of Alfa-1, a simulated computer with educational purposes. The DEVS formalism was used to attack the complexity of the design, allowing the definition of individual components that can be lately integrated into a modelling hierarchy. The tool is designed for the use in Computer Architecture and Organization courses. Its goal is allowing the students to acquire some practice in the design and implementation of hardware components by using simulation.

1. Introduction

The theoretical study of computer architecture and organization usually give the students an incomplete and sometime erroneous view of how a computer system works. Computer organization bibliography (for instance, [1, 2, 3]) usually emphasizes the basic behavior of a computer system. The lack of practical experience can make the underlying complexity of the subsystems and their interaction not to be completely understood.

The main problems are related with the existence of several layers to be studied. The assembly language, instruction set microprogramming and digital logic levels are usually considered. The introduction of higher levels (programming languages, Operating Systems) makes the task even more difficult.

The inter-level interaction can make the students to loose the understanding of the system operation as a whole. In addition, the detailed behavior for each of the subsystems can be complex to analyze. The existence of hardware or software tools with educational purposes in this area is reduced, making difficult to make the concepts clear through practice.

In [4] a set of tools devoted to simulate the components of a computer were presented. These tools cannot be applied if several levels need to be integrated. Furthermore, detailed specification of the components cannot be achieved easily. To avoid these problems, the simulated computer was completely redesigned using a formal approach. The DEVS formalism [5] was used, due to the hierarchical and discrete events nature of the problem.

A real system modeled using DEVS can be described as a set of behavioral (atomic) and structural (coupled) submodels. The models can be hierarchically integrated, allowing model reuse. This improves the security of the simulations, reduces testing time and enhances productivity. A DEVS atomic model is described by:

$$M = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D >.$$

X: input events set;
S: state set;
Y: output events set;
δ_{int}: $S \rightarrow S$, internal transition function;
δ_{ext}: $Q \times X \rightarrow S$, external transition function,
$Q = \{ (s, e) / s \in S$, and $e \in [0, D(s)]\}$;
λ: $S \rightarrow Y$, output function; and
D: $S \rightarrow \boldsymbol{R}_0^+$, elapsed time function.

This work was partially supported by USENIX,
ANPCYT Project 11-04460 and
UBACYT Project JW10.

Each model uses input/output ports to communicate with others. The input external events are received in input ports, and the model specification defines the behavior under such inputs. The internal events produce state changes, whose results are spread through the output ports.

DEVS coupled models are defined by:

$$CM = < X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, select >$$

X is the set of input events;
Y is the set of output events;
D is an index of components, and $\forall i \in D$,
$\mathbf{M_i}$ is a basic DEVS model;
$\mathbf{I_i}$ are the influencees of model i, and $\forall j \in I_i$
$\mathbf{Z_{ij}}$: $Y_i \rightarrow X_j$ is the i to j translation function.
Finally, **select** is the tie-break selector.

The tool GAD [6] implements the theoretical concepts of the DEVS formalism. This tool was used as a basis to develop the simulated computer, allowing to experiment with the formal approach defined by the DEVS formalism. The results obtained will be shown in the following sections.

2. CPU architecture description

The model's architecture is mainly based in the specification of the *Integer Unit* of the SPARC processor (Sun Microsystems). The actual design of this processor was used, but the instruction set and the memory management were simplified due to educational purposes. The Figure 9 in the Appendix presents a sketch of the processor's organization.

The design of the memory is flat (neither segmentation nor pagination are included) and multiprogramming is not supported. The implementation includes two registers: *Base* and *Size*, which defines the memory space of the program.

The processor is provided with 520 integer registers, divided in three classes. Eight of them are global, and the remaining 512 are divided in windows of 24 registers each. These include input, output and local registers for each procedure. When a procedure starts, 16 registers are reserved (8 local and 8 for output), and the 8 output records of the calling procedure are used as inputs.

A specialized 5-bit register, called *CWP* (Circular Window Pointer), marks the active window into the register ring. Each time that a new procedure is started, *CWP* is decremented. The 32-bit *WIM* register (Window Invalid Mask, one bit per window) avoids the superposition with a window in use by another procedure. When *CWP* is decremented, the hardware verifies if *WIM* is on for the new window. In that case, an interrupt is raised. The interrupt service routine saves the content of the window, which will be overwritten. Usually, *WIM* only has one bit in 1, marking the oldest window. When that window is reached, the *WIM* rotates one unit.

Besides the general use registers, the following registers can be used:

- **Y**: used by the product and division operations.
- **TBR** (Trap Base Register): it points the memory address where the trap routine starts.
- **BASE and SIZE:** BASE points to the lower address that the program can access, and SIZE stores the program size.
- **PSR** (Processor Status Register): stores the present program status.

The processor uses two program counters: PC (containing the address of the next instruction), and nPC (Next Program Counter, storing the address for the following PC). Each instruction cycle finishes by copying the nPC to the PC, and adding 4 to the nPC. If the instruction is a branch, nPC is assigned to PC, and nPC is updated with the jump address.

3. Architecture implementation

The basic components were defined as DEVS models, which have been coupled into complex ones. Two implementations have been defined at present. The first reproduced the basic behavior of each component, coded as transition functions. Then, the digital logic level was implemented. The basic building blocks were developed as atomic models, coupling them using digital logic concepts.

3.1. Inc/Dec

This component is used to update the *CWP*. It increments or decrements a 5-bit value received through the lines OP0-OP4. Then, the output is transmitted through RES0-RES4 (OP0 and RES0 are the MSB). This atomic model is defined by:

INC/DEC = < X, S, Y, δ_{int}, δ_{ext}, λ, D >.

X $\in$ {1,...,32} $\cup$ {0,1} ;

S $\in$ { OP } $\cup$ { FCOD } $\cup$ { RES } ;

Y $\in$ {0,1} ;

The following figure shows the implementation of the transition functions using the tool.

```
Model &IncDec::externalFunction( const
ExternalMessage &msg ) {

 switch (msg.port()) {
 // When x is received in the port y;
   case OP0:  _OP[0] = (int) msg.value();
   case OP1:  _OP[1] = (int) msg.value();
   case OP2:  _OP[2] = (int) msg.value();
   case OP3:  _OP[3] = (int) msg.value();
   case OP4:  _OP[4] = (int) msg.value();
   case FCOD: _FCOD  = (int) msg.value();
  }

  if( _FCOD == 1 ) // increment
    _RES[0]=(_OP[0]*16+_OP[1]*8+_OP[2]*4
        +_OP[3]*2+_OP[4])+1) % 32;
  else // decrement
    _RES[0]=((_OP[0]*16+_OP[1]*8+_OP[2]*4+
        _OP[3]*2+_OP[4]*0)-1) %32;

  for (int i=4; i>=1; i--){ // to binary
   _RES[i]=_RES[0]%2; _RES[0]=_RES[0]/2;
}

   this->holdIn(active, preparationTime);
       return *this;
   }

Model &IncDec::internalFunction( const
InternalMessage & ) {
       this->passivate();
        return *this ;
}

Model &IncDec::outputFunction( const In-
ternalMessage &msg )  {

 for (int i=0; i<5; i++) {
 // If OP0..4 is different of last output
  if (_RES[i]!=_OLD[i]) { // send OP0..4
   for (int j=0; j<5; j++)  {
     sendOutput(msg.time(),RESj, _RES[j]);
     _OLD[j]=_RES[j]; }
  }
     return *this ;
}
```

Figure 1. INC/DEC [7].

The *external transition* function (δ_{ext}) receives five operands as inputs, together with a function code. According to this code, the parameter is incremented or decremented. After, the model keeps the present value during a preparation time. The *output* function (λ) is activated to see if the circuit has changed its state. In that case, it transmits its present value. Then, the *internal transition* function (δ_{int}) passivates the model. The D function is managed by the *hold_in* and *passivate* macros used in the model.

3. 2. RegGlob

This model defines the behavior of the global registers. It keeps the contents of the 8 global registers, allowing the read/write operations on them.

```
Model &Regglob::externalFunction( const
ExternalMessage &msg ) {
  switch (msg.port()) {
    case cen: bcen = (int)msg.value();
    case reset:breset = (int)msg.value();
   }

   if( msg.port() == cin+i )
     in[i]=(int)msg.value();

   if( msg.port() == asel+i ) {
     sela[i]=(int) msg.value();
     selecta=sela[0]+2*sela[1]+4*sela[2];
   }

   if( msg.port() == bsel+i ) {
     selb[i]=(int) msg.value();
     selectb=selb[0]+2*selb[1]+4*selb[2];
   }

   if( msg.port() == csel+i ) {
     selc[i]=(int) msg.value();
     selectc=selc[0]+2*selc[1]+4*selc[2];
   }
   this->holdIn ( active, delay );
   return *this;
}

Model &Regglob::internalFunction( const
InternalMessage &msg )  {

   if (bcen && (selectc>0))
    for (int i=0; i<32; i++)
       g[(selectc*32)+i]=in[i];
   if (breset)
     for (int i=0; i<255; i++)g[i]=0;

   this->passivate();
   return *this ;
}

Model &Regglob::outputFunction( const In-
ternalMessage &msg )  {

   if (olda[i]!=g[selecta*32+i]) {
     this->sendOutput(msg.time(), aout+i,
        g[selecta*32+i]);
     olda[i]=g[selecta*32+i];  }

   if (oldb[i]!=g[selectb*32+i]) {
      this->sendOutput(msg.time(), bout+i,
        g[selectb*32+i]);
      oldb[i]=g[selectb*32+i];  }

   return *this ;
}
```

Figure 2. RegGlob [8].

A sketch of this model is shown in the Appendix. It uses three select lines (*asel, bsel, csel*) to choose the two output registers and the register to be modified. An array of 32 integers (IN) keeps the present input values. The boolean variable *cen* stores the state of the C enable line. The *reset* variable stores the present state of the reset line. Finally, *Aout* and *Bout* stores the previous status of the A and B output lines.

The external transition function is in charge to receive a pulse through the input values. If the input signal goes to one of the selectors, the number of the chosen register is computed. The next internal event is scheduled with a predefined delay. If an external event arrives before the end of the delay, it is restarted.

The internal transition function sees if the internal value *reset* is activated. In that case, it clears the contents of every register. If the *cen* variable is active and the register in the selector C is not zero, the value of the register is updated with the input values. After, the model passivates. The output function just sends the contents specified by the selectors A and B through the AOUT and BOUT ports.

4. The digital logic level

The abstraction level of several models was detailed with educational purposes. In this way, the digital logic level of the integrating circuits can be analyzed. These models were built using atomic models representing the basic logical gates (AND, OR, NOT, XOR). After, they were integrated in complex models by coupling them. Two of the implemented models will be explained following.

4.1. CMP model

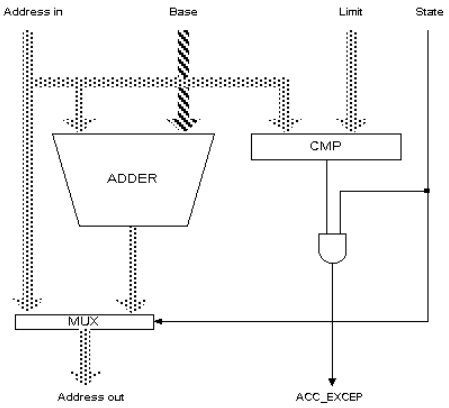

Figure 3. Sketch of the Address Unit.

The CMP register is part of the Address Unit. It must detect addresses falling out of the program boundaries. The model receives two inputs (through the registers *OPA* and *OPB*), and returns the signal *EQ* if both values are equal, or *LW* if A is lower than B.

The model is composed of several one-bit comparators, defined as atomic models. N-bits comparators are generated by coupling N one-bit comparators. The following figure shows the basic components of this building block:

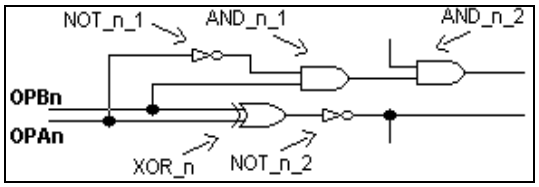

Figure 4. One-bit comparator [9].

This model is formally described by:

$$CM = < X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, select >$$

$\boldsymbol{X}$ = {OPAn, OPBn / OPAn, OPBn ∈ {0,1} };
$\boldsymbol{Y}$ = {EQ, LW / EQ, LW ∈ {0,1} };
D = {NOT_n_1, NOT_n_2, XOR_n, AND_n_1, AND_n_2 }; where each is an atomic defining the corresponding building block;
$\mathbf{I}_{\mathbf{NOT\_n\_1}}$ = { AND_n_1 };
$\mathbf{I}_{\mathbf{XOR\_n}}$ = {NOT_n_2 };
$\mathbf{I}_{\mathbf{NOT\_n\_2}}$ = { Self };
$\mathbf{I}_{\mathbf{AND\_n\_2}}$ = { Self };
$\mathbf{I}_{\mathbf{AND\_n\_1}}$ = { AND_n_2 };
$\mathbf{I}_{\mathbf{self}}$ = { Self, NOT_n_1, AND_n_1, XOR_n};
select = **D**; and
$\mathbf{Z}_{\mathbf{ij}}$ is built using I, as described earlier.

The definition of this coupled model using the tool is presented in the following figure:

```
[top]
components : NOT_n_1@NOT NOT_n_2@NOT
        XOR_n@XOR AND_n_1@AND AND_n_2@AND
in : OPAn OPBn
out : LW EQ

Link : OPAn in@NOT_n_1
Link : OPBn ina@XOR_n
Link : OPAn inb@XOR_n
Link : OPBn inb@AND_n_1
Link : out@NOT_n_1 ina@AND_n_1
Link : out@XOR_n in@NOT_n_2
Link : out@AND_n_2 EQ
Link : out@NOT_n_2 LW
```

Figure 5. CMP coupled model [9].

4.2. Chip Selector

The Chip Selector (CS) circuit is devoted to determine if an address is between two others. The model receives a 32-bit address and an Address Strobe (AS), and it returns the Chip Selector value, computed as: CS = AS $\wedge$ A $\in$ Range. The range is a set of addresses that act as a maximum and minimum addresses for the chip selector. The AS is devoted to activate the circuit. A sketch of the circuit is found in the Appendix.

```
[top]
components: MASMAX@MAS MASMIN@MAS CMPA@CMP
CMPB@CMP and1@AND and2@AND or@OR not@NOT
in : A31 A30 A29 A28 A27 A26 A25 A24 A23
A22 A21 A20 ... A4 A3 A2 A1 A0 AS
out : CS

Link:   A31   OPA31@CMPA   A31   OPA31@CMPB
Link: A30 OPA30@CMPA A30 OPA30@CMPB
...
Link:    A1    OPA1@CMPA    A1    OPA1@CMPB
Link: A0 OPA0@CMPA A0 OPA0@CMPB

Link: out31@MASMAX OPB31@CMPA out31@MASMIN
OPB31@CMPB
Link: out30@MASMAX OPB30@CMPA out30@MASMIN
OPB30@CMPB
...
Link:  out0@MASMAX  OPB0@CMPA  out0@MASMIN
OPB0@CMPB
Link: AS ina@and2
Link: eq@CMPA ina@or  lw@CMPA  inb@or
Link: lw@CMPB in@not
Link: out@or ina@and1  out@not inb@and1
Link: out@and1 inb@and2
Link: out@and2 CS
```

Figure 6. CS coupled model [9].

A MASK specialized model was defined. The function of this model is to provide two 32-bit sets containing the boundaries of the set to be compared. The two input addresses (CMP A, CMP B) are compared with these boundaries (MAX Mask, MIN Mask). These models are comparators as those defined in the previous section.

The result obtained is transmitted through the ports LW and EQ for each of the comparators. Both outputs are ORed for the first register (as we are interested to see if CMP A $\leq$ MAX). After, the LW output of the second register is inverted (as we are interested to see if CMP B $\geq$ MIN). If the circuit is enabled, the result obtained is transmitted.

5. Simulation results

The present section is devoted to show some of the results obtained when the models previously presented are simulated. In the first case, we show the results obtained simulating the INC/DEC model.

INPUT	OUTPUT
00:00:00:00 OP0 1	00:00:05:000 res0 1
00:00:05:00 OP1 0	00:00:05:000 res1 0
00:00:10:00 OP2 1	00:00:05:000 res2 0
00:00:15:00 OP3 0	00:00:05:000 res3 0
00:00:20:00 OP4 0	00:00:05:000 res4 0
00:00:25:00 FCOD 1	00:00:15:000 res0 1
	00:00:15:000 res1 0
	00:00:15:000 res2 1
	00:00:15:000 res3 0
	00:00:15:000 res4 0
	00:00:30:000 res0 1
	00:00:30:000 res1 0
	00:00:30:000 res2 1
	00:00:30:000 res3 0
	00:00:30:000 res4 1

Figure 7. Inputs and Outputs for the INC/DEC model.

This example shows how a value of 20 is incremented by the circuit. The first step consists in the giving an initial value. When the simulation is started, the components have initial value of zero. Therefore, the reception of the first event (OP0 = 1 at 00:00:00:00) will generate an output when the phase changes (00:00:05:00).

As the second input does not generate changes in the model, no output can be detected. In the simulated time 10, a new input is inserted through the port OP2. Then, this value is changed and the output generated. As the preparation time for the circuit is 5 time units, the outputs are produced at simulated time 15. The following 2 inputs are not registered. The last one changes the register by inserting the value through the FCOD port. Therefore, the register is incremented.

The second example shows the execution obtained of the RegGlob model under different inputs. At the instant 0, the C enable line is activated. The register 4 is selected (*csel2=1*), and the number FFFFFFFFh is used as input (*cin0 = ... = cin31=1*). After, in 00:00:01:00, the register 2 is selected (*csel2=0 and csel1=1*), and the number 55555555h is input (*cin0=cin2=cin4...=cin30=1,* and *cin1= cin3 = cin5 =... =cin30=1*).

At 00:00:02:00, C Enable is deactivated. The register 4 is selected for A, and the register 2 is chosen for B. As a result, the values previously loaded are read (that is, FFFFFFFFh in A, and 55555555h in B).

INPUT	OUTPUT
00:00:00:00 cen 1 00:00:00:00 csel2 1 00:00:00:00 cin0 1 ... 00:00:00:00 cin31 1	
00:00:01:00 csel2 0 00:00:01:00 csel1 1 00:00:01:00 cin1 0 00:00:01:00 cin3 0 00:00:01:00 cin5 0 ... 00:00:01:00 cin29 0 00:00:01:00 cin31 0	
00:00:02:00 cen 0 00:00:02:00 asel2 1 00:00:02:00 bsel1 1	00:00:02:010 aout0 1 ... 00:00:02:010 aout31 1 00:00:02:010 bout0 1 00:00:02:010 bout2 1 00:00:02:010 bout4 1 ... 00:00:02:010 bout30 1
00:00:04:00 reset 1	
00:00:05:00 asel2 1 00:00:05:00 asel1 0	00:00:05:010 aout0 0 00:00:05:010 aout2 0 ... 00:00:05:010 aout28 0 00:00:05:010 aout30 0

Figure 8. Inputs/Outputs of RegGlob.

After, Reset is activated. When the circuit is activated to read the register 4 at 00:00:05:00, it returns the value 00000000h.

6. Conclusion

We have presented the use of a Discrete Event formalism and a simulation environment to build a set of tools to simulate a simple computer. The tools can be used in Computer Organization courses to analyze and understand the basic behavior of the different levels of a computer system. The interaction between levels can be studied, and experimental evaluation of the system can be done.

A basic instruction set was considered, based on the SPARC architecture, allowing to study the main features of this processor, analyzing features not existing in simpler processors. Several basic circuits were also implemented, allowing to build the computer by using them. At present, both levels are being connected by a Control Unit.

The use of this set of tools allowed the students to obtain a complete understanding of the computer organization. At present, the set of tools is being completed by including an input/output subsystem. In addition, an assembler and linker for the tool are being built.

These tools are public domain, and at present they can be obtained at the author's URL.

References

[1] Stallings, W. *Computer Organization and Architecture.* Macmillan, New York. 4th Edition. 1996.

[2] Tanenbaum, A., *Structured Computer Organization*, 3rd edition, Prentice Hall, New Jersey, 1990.

[3] Hennessy, J. and Patterson, D. *Computer Architecture: a quantitative approach*. Prentice Hall International. 1994.

[4] Wainer, G. "ALFA-0: a simulated computer as an educational tool for Computer Organization". *Proceedings of IASTED Applied Modelling and Simulation 1998*. Hawaii, USA. 1998.

[5] Zeigler, B. *Theory of modeling and simulation*. Wiley, 1976.

[6] Barylko, A.; Beyoglonián, J. and Wainer, G. "GAD: a General Application DEVS environment". *Proceedings of IASTED Applied Modelling and Simulation 1998*. Hawaii, USA 1998.

[7] Barletta, A.; Enrique, S.; Rubinstein, D. “Definition of ALU components for the Alfa-1 simulated processor”. (in Spanish). *Internal report. Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires.* 1998.

[8] Barrionuevo, J.; Calvo, A. and Corvetto, A. “Definition of global registers components for the Alfa-1 simulated processor”. (in Spanish). *Internal report. Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires.* 1998.

[9] Petronio, F. “Defining the digital logic level of components for the Alfa-1”. (in Spanish). *Internal report. Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires*. 1999.

APPENDIX

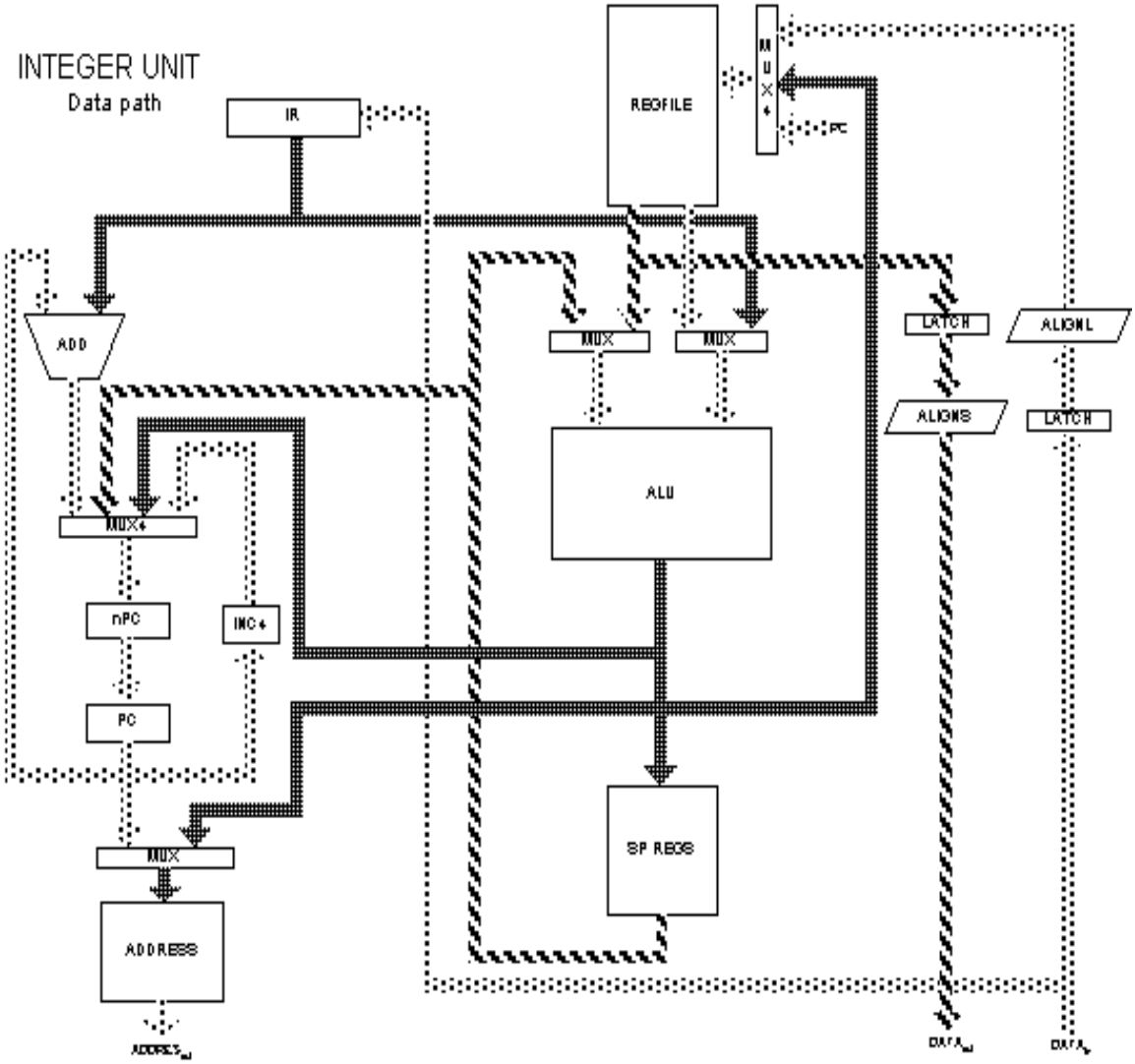

Figure 9. Data path of the simulated processor.

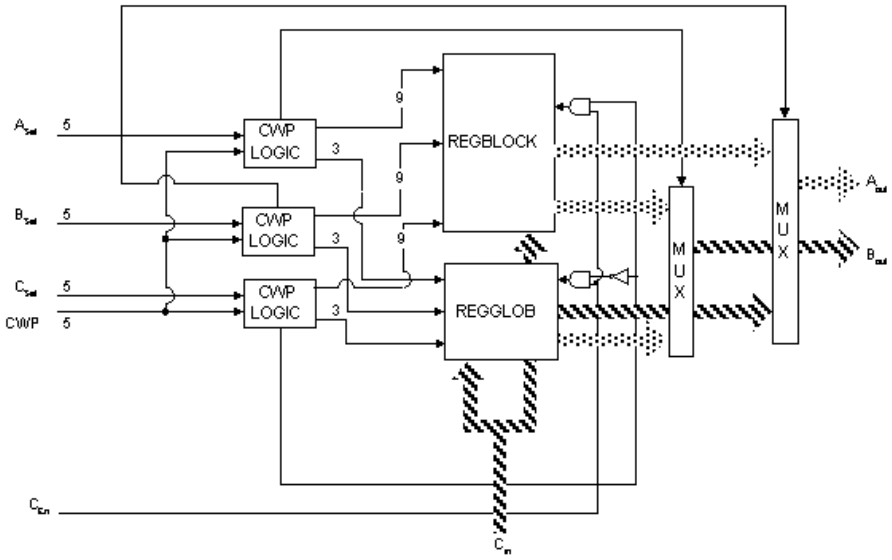

Figure 10. Architecture of the processor's registers.

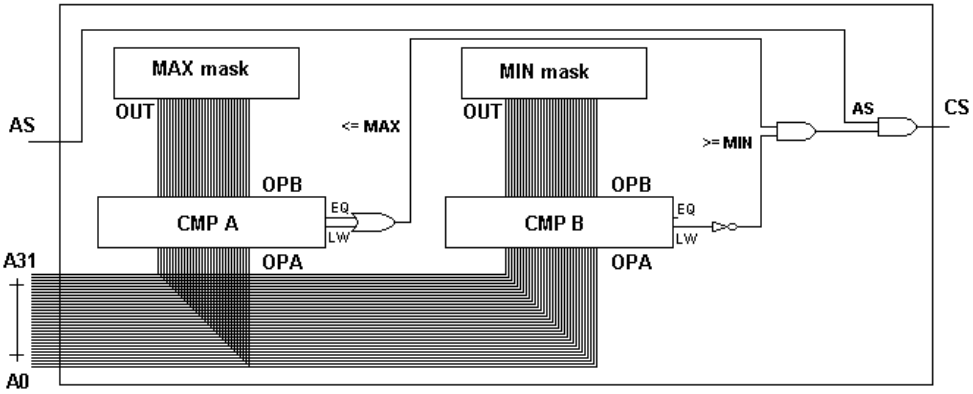

Figure 11. Sketch of the Chip Selector [9].