EXPLORING THE REGISTER RENAMING DESIGN SPACE USING THE SUPERSIM SIMULATOR

A. Mišev, M. Gušev

Institute of Informatics, Faculty of Natural Sciences and Mathematics,
Sts. Cyril and Methodius University,
Arhimedova bb, PO BOX 162, Skopje, Macedonia
{infolab, marjan}@ii.edu.mk

Abstract: Usage of simulators in the process of processor development is very common thing. Simulators are used mainly in designing new techniques to improve performances. Main purpose of all these techniques is extracting more parallelism by resolving dependencies. Register renaming is a mechanism for coping with false data dependencies. Exploring the design space for register renaming gives plenty of opportunities to improve the overall performance.

Keywords: register renaming, superscalar processor, out of order, simulation.

1. Introduction

Superscalar processors, along with VLIW processors exploit parallelism at instruction level. The most powerful mean used by superscalars is out-of-order instruction execution. Independent instructions are executed out of their program order to achieve more parallelism. Several factors imply on the amount of concurrently executing instructions. Data dependencies, along with control and resource dependencies, have the biggest influence.

The superscalar instruction processing can be divided into several specific tasks. Some of them are pre-decoding, decoding, instruction issue, execution and preservation of the sequential consistency of execution.

The most complicated of them is the superscalar out-of-order instruction issue. Advanced mechanisms for exploiting more ILP are incorporated in this phase. They include shelving and register renaming as mechanisms for resolving data dependencies.

Since the development of new techniques requires a lot of experimental work to be done, using simulators for research seems to be the best choice. Several simulators of ILP processors are available, each with different possibilities.

2. Register renaming design space

Register renaming represents a mechanism for coping with false data dependencies. Tjaden and Flynn [13] introduce this technique in 1970. Keller [9] gave the notion of register renaming in 1975. Register renaming is applicable in two- or three-address format register-register or register-immediate instructions, assuming one or two source registers and one target register. The main idea of register renaming is holding the results of the instructions in temporal locations instead of writing it directly to the destination registers. This enables the instruction to execute out-of-order and to write the results to the architectural registers in-order.

Register renaming can be static or dynamic. The compiler performs static renaming at compile time. Dynamic renaming is performed by special hardware during instruction execution. Dynamic register renaming is considered in this work.

The design space of the register renaming technique consists of the scope of the renaming, the layout of the rename registers, the mapping of the rename registers and the rename rate.

The scope of the renaming can be partial and full, depending on whether some or all of the instructions use renaming. All of the current processors use full renaming.

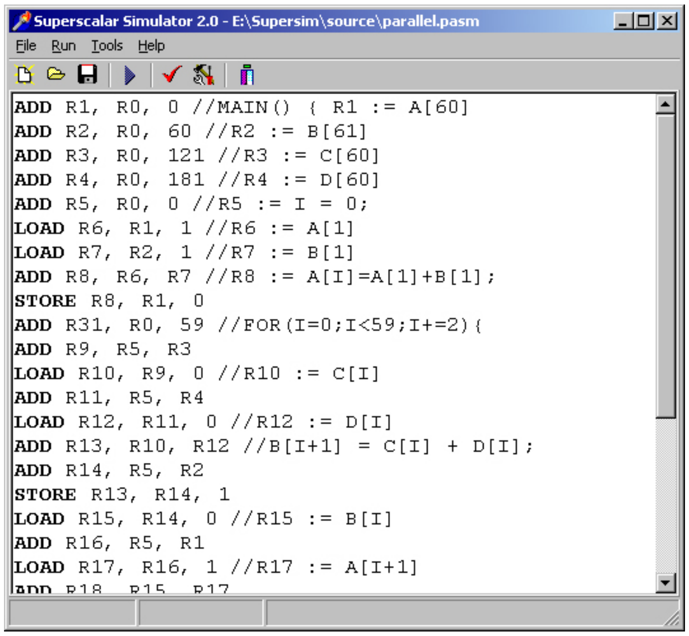
According to the layout of the rename registers, there are several different approaches. The first approach uses the reorder buffer to hold the temporal values. This is the most natural implementation, since the reorder buffer is responsible for in-order instruction retirement. But this approach has some disadvantages. The main disadvantage is the need of read and writes ports to the ROB, similar to the ports of the register files and it induces more complexity to the operand fetch phase. The other approach is using a single register file for the architectural and renamed registers. The third approach is to use a separate register file for renaming. This approach is implemented in our simulation tool. For every issued instruction that has a destination register, a new mapping is made for the destination register to a new physical register. An important aspect of the renaming mechanism design is the reuse of the registers. It specifies the time between the allocation of a physical register and its freeing when no longer needed.

There are two possibilities for register mapping. The first one is to use indexed mapping table. In this case, each register can have only one rename. The other, more commonly used way is to use associative mapping table. Every issued instruction with destination register is assigned a new rename register through the mapping table. This mapping remains until the instruction commits or cancels. In the first case, the value held in the rename register is written into the architectural register, altering the processors state. The rename register is freed for new mapping. In the second case, the instruction is not allowed to alter the processors state, so the value is discarded and the rename register is freed. The following instructions use the mapping table to reference the rename register as their source register, instead of the architectural one. When register renaming is used, values written to architectural registers are always held valid. Only the rename register file needs additional mechanisms for marking the validity of the contents of its registers.

The rename rate represents the maximum number of renames that can be made per cycle. It is usually equal to the issue rate, since the renaming is performed in the issue phase.

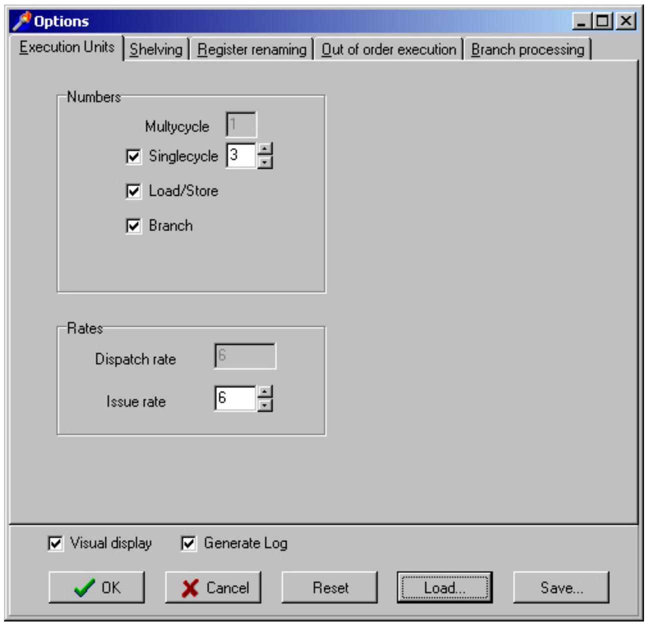
3. SuperSim simulator

The basic considerations for designing the SuperSim Simulator were taken from the design space concept given by Sima et al [11], using similar experience of [1]. The previous versions of the simulator are covered in [5].


Superscalar Simulator 2.0 - E:\Supersim\source\parallel.pasm
File
Run
Tools
Help

```
ADD R1, R0, 0 //MAIN() { R1 := A[60]
ADD R2, R0, 60 //R2 := B[61]
ADD R3, R0, 121 //R3 := C[60]
ADD R4, R0, 181 //R4 := D[60]
ADD R5, R0, 0 //R5 := I = 0;
LOAD R6, R1, 1 //R6 := A[1]
LOAD R7, R2, 1 //R7 := B[1]
ADD R8, R6, R7 //R8 := A[I]=A[1]+B[1];
STORE R8, R1, 0
ADD R31, R0, 59 //FOR(I=0;I<59;I+=2){
ADD R9, R5, R3
LOAD R10, R9, 0 //R10 := C[I]
ADD R11, R5, R4
LOAD R12, R11, 0 //R12 := D[I]
ADD R13, R10, R12 //B[I+1] = C[I] + D[I];
ADD R14, R5, R2
STORE R13, R14, 1
LOAD R15, R14, 0 //R15 := B[I]
ADD R16, R5, R1
LOAD R17, R16, 1 //R17 := A[I+1]
ADD R18, R15, R17
```

Figure 1: The code entry window



Numbers

Multycycle 1

[✓] Singlecycle 3

[✓] Load/Store

[✓] Branch

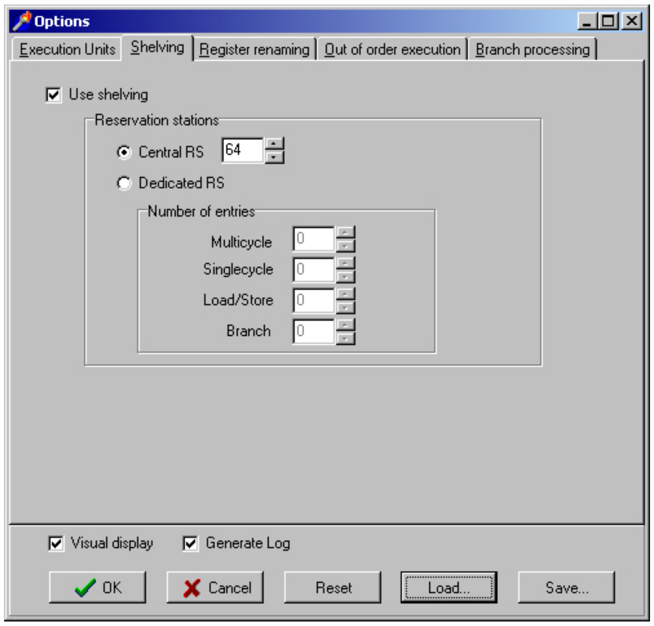
Rates

Dispatch rate 6

Issue rate 6

[✓] Visual display [✓] Generate Log

Figure 2: The options window



[✓] Use shelving

Reservation stations

● Central RS 64

○ Dedicated RS

Number of entries

Multicycle 0

Singlecycle 0

Load/Store 0

Branch 0

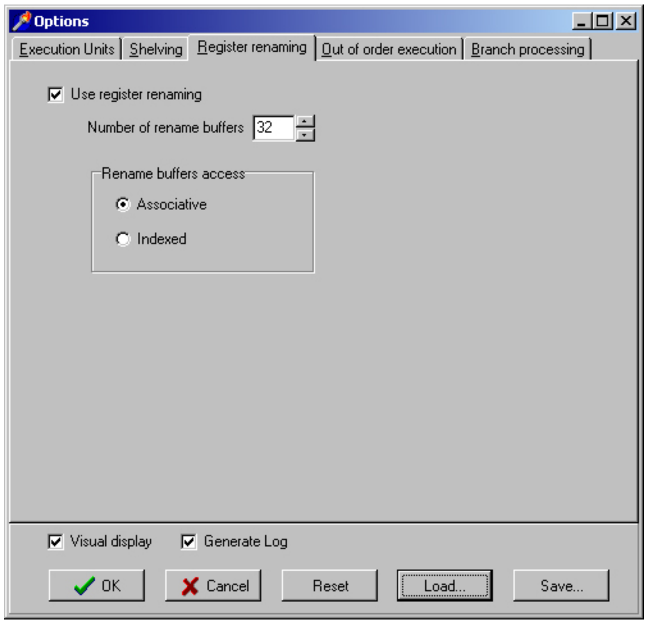
[✓] Visual display [✓] Generate Log

Figure 3: Shelving options

The code editor window, shown in Fig. 1, enables the user to write its own custom code, using the pseudo assembler. Options available on this window include syntax checking with indication of possible errors and standard file management.

Especially important is the configuration option, which defines the simulated execution environment. It consists of several major parts, each represented with a tab, as shown in Fig. 2. The configuration enables choosing the number and the type of the execution units. The maximum number of execution units is 6, and the minimum is 1. Supported units are: 1 multi cycle unit, for execution of multi cycle integer operations, like division or multiplication; up to 3 single cycle integer units, for execution of simple integer arithmetic; 1 load/store unit for address calculation of the memory transfer instructions and 1 branch unit for calculation of the branch target addresses.

Only the multi cycle unit is mandatory, while the others can be added or removed. If a special unit is not used, for example the load/store unit, the multi cycle unit performs the operations. The issue rate can also be configured on this tab, varying from 1 up to the total number of units used. The second tab of the configuration window, shown in Fig. 3, covers the use of shelving. When shelving is used, the user can select between central or dedicated reservation stations. For each station used, the number of entries can also be configured.



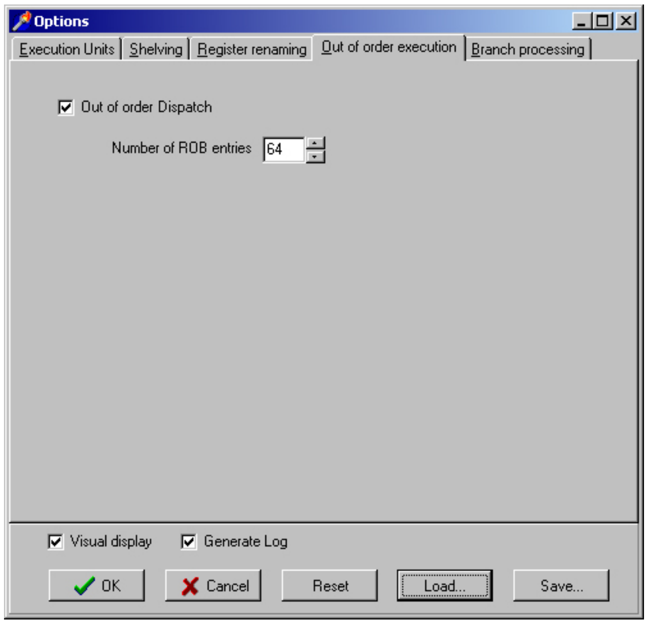
☑ Use register renaming

Number of rename buffers: 32

Rename buffers access: [✓ Associative] Indexed

☑ Visual display ☑ Generate Log

Figure 4: Register renaming options

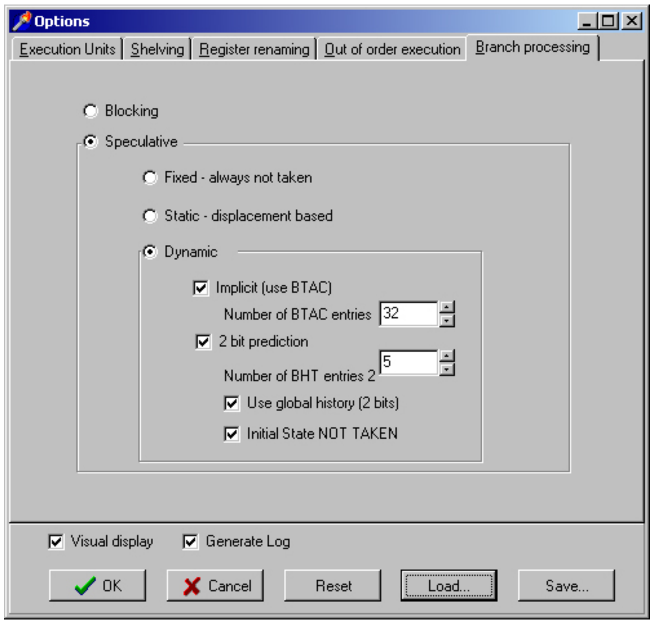


[✓] Out of order Dispatch

Number of ROB entries 64

[✓] Visual display [✓] Generate Log

Figure 5: Out-of-order options



○ Blocking

● Speculative

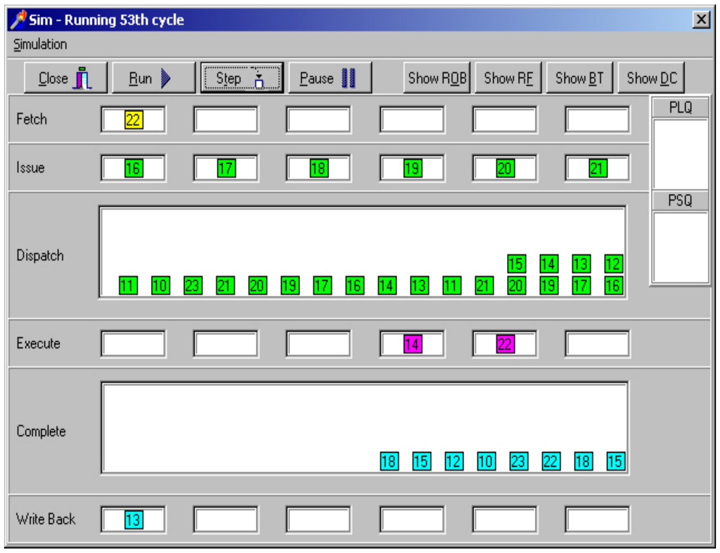
- ○ Fixed - always not taken
- ○ Static - displacement based
- ● Dynamic
  - [✓] Implicit (use BTAC)
    - Number of BTAC entries 32
  - [✓] 2 bit prediction
    - Number of BHT entries 2 5
    - [✓] Use global history (2 bits)
    - [✓] Initial State NOT TAKEN

[✓] Visual display [✓] Generate Log

Figure 6: Branch processing options

The next tab, Fig. 4, is used for configuring the register renaming options of the simulator. If renaming is used, the number of rename buffers can be selected. Additionally, the access method for the renamed registers can be chosen from indexed or associative. The Out of order tab, Fig. 5, enables the using of the out of order issue and dispatch. On the same tab, the user can adjust the number of entries in the Reorder Buffer (ROB). The final configuration tab covers the branch processing used in the simulation, as shown in Fig. 6. It can be blocking or speculative. When using speculative branch prediction, three modes are available: fixed, static and dynamic. The dynamic branch processing can be configured to use BTAC, BHT or both. It can also use global 2-bit history, for better prediction. Other options available are turning on and off the visual simulation, which can increase performance and tuning on and off the logging option. When visualization is disabled, the number of clock cycles simulated per second is 7-10 times bigger. The selected configuration can be saved into a file for later reuse, or loaded from one.

The runtime environment greatly depends on the selected configuration. When full configuration is used, it looks like in Fig. 7. The top part consists of some command buttons and the rest of the window is divided into separate parts for each stage of the pipeline. Mandatory stages are Fetch, Issue, Execute and Write-back, while the other two, Dispatch and Complete are shown only if shelving and out-of-order execution are used, respectively. For each stage, a container represents the appropriate tables and/or buffers that hold the current instructions. In the upper left part, two separate containers represent the pending load and store queues. The ROB window, shown in Fig. 8 is used for monitoring the work of the reorder buffer. It has an entry for each instruction that has been issued and has not completed yet. Since the ROB is designed as a circular buffer, at also shows the head and the tail pointer in the buffer.

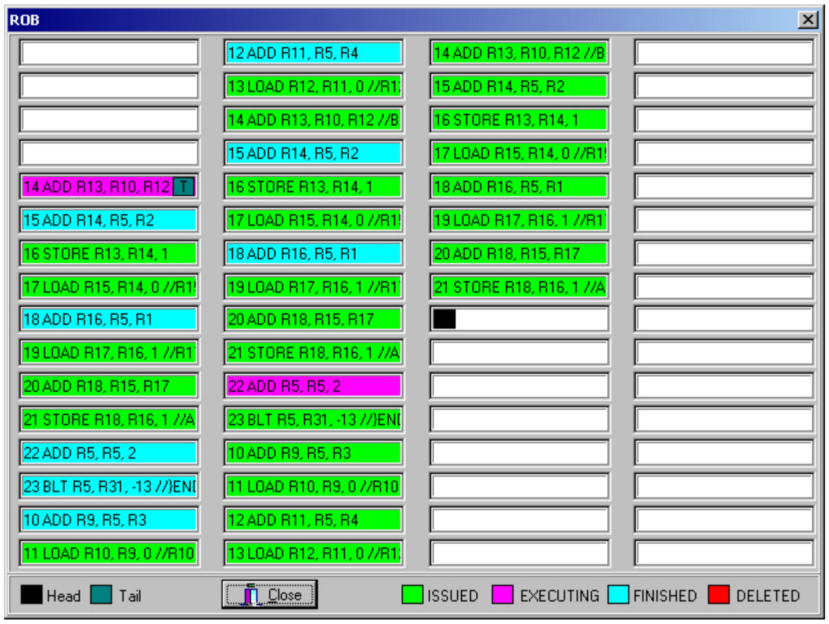


Fetch	22					
Issue	16	17	18	19	20	21
Execute				14	22	
Write Back	13					

Dispatch: 15 14 13 12 / 11 10 23 21 20 19 17 16 14 13 11 21 20 19 17 16

Complete: 18 15 12 10 23 22 18 15

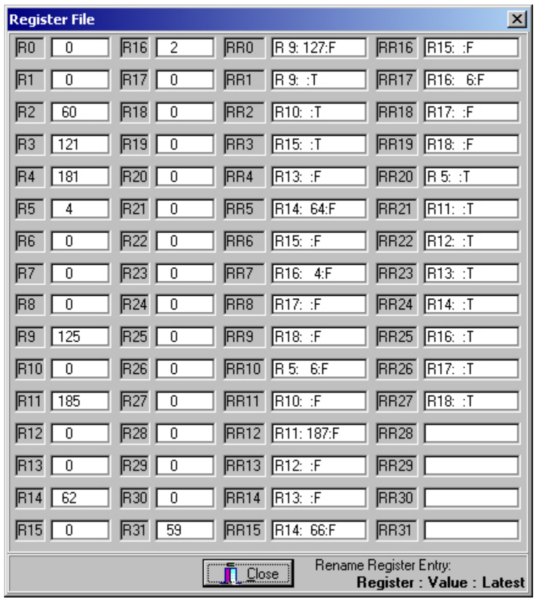
Figure 7: The runtime window



	12 ADD R11, R5, R4	14 ADD R13, R10, R12 //B	
	13 LOAD R12, R11, 0 //R1:	15 ADD R14, R5, R2	
	14 ADD R13, R10, R12 //B	16 STORE R13, R14, 1	
	15 ADD R14, R5, R2	17 LOAD R15, R14, 0 //R1!	
14 ADD R13, R10, R12 T	16 STORE R13, R14, 1	18 ADD R16, R5, R1	
15 ADD R14, R5, R2	17 LOAD R15, R14, 0 //R1!	19 LOAD R17, R16, 1 //R1	
16 STORE R13, R14, 1	18 ADD R16, R5, R1	20 ADD R18, R15, R17	
17 LOAD R15, R14, 0 //R1!	19 LOAD R17, R16, 1 //R1	21 STORE R18, R16, 1 //A	
18 ADD R16, R5, R1	20 ADD R18, R15, R17	■	
19 LOAD R17, R16, 1 //R1	21 STORE R18, R16, 1 //A		
20 ADD R18, R15, R17	22 ADD R5, R5, 2		
21 STORE R18, R16, 1 //A	23 BLT R5, R31, -13 //ENI		
22 ADD R5, R5, 2	10 ADD R9, R5, R3		
23 BLT R5, R31, -13 //ENI	11 LOAD R10, R9, 0 //R10		
10 ADD R9, R5, R3	12 ADD R11, R5, R4		
11 LOAD R10, R9, 0 //R10	13 LOAD R12, R11, 0 //R1:		

■ Head ▨ Tail 🟩 ISSUED 🟪 EXECUTING 🟦 FINISHED 🟥 DELETED

Figure 8: The ROB window

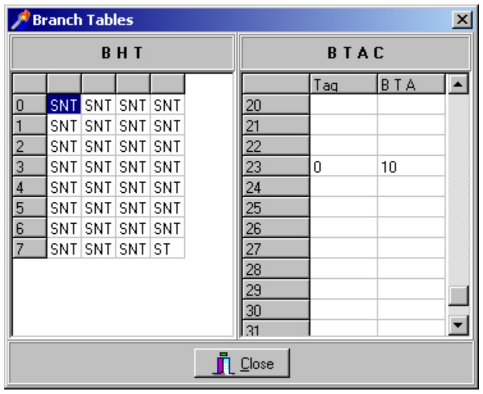


R0	0	R16	2	RR0	R 9: 127:F	RR16	R15: :F
R1	0	R17	0	RR1	R 9: :T	RR17	R16: 6:F
R2	60	R18	0	RR2	R10: :T	RR18	R17: :F
R3	121	R19	0	RR3	R15: :T	RR19	R18: :F
R4	181	R20	0	RR4	R13: :F	RR20	R 5: :T
R5	4	R21	0	RR5	R14: 64:F	RR21	R11: :T
R6	0	R22	0	RR6	R15: :F	RR22	R12: :T
R7	0	R23	0	RR7	R16: 4:F	RR23	R13: :T
R8	0	R24	0	RR8	R17: :F	RR24	R14: :T
R9	125	R25	0	RR9	R18: :F	RR25	R16: :T
R10	0	R26	0	RR10	R 5: 6:F	RR26	R17: :T
R11	185	R27	0	RR11	R10: :F	RR27	R18: :T
R12	0	R28	0	RR12	R11: 187:F	RR28	
R13	0	R29	0	RR13	R12: :F	RR29	
R14	62	R30	0	RR14	R13: :F	RR30	
R15	0	R31	59	RR15	R14: 66:F	RR31	

Rename Register Entry:
Register : Value : Latest

Figure 9: The Register file window

The registry file window, Fig. 9, shows the state of both the architectural and the rename registers. On the left of the window, architectural registers are shown. For each rename register, there are three parameters shown: the number of the architectural register that is mapped to this rename register, the value (if calculated yet) and the latest bit. The branch tables' window, Fig. 10, is used for monitoring the state of the branch prediction tables. Depending on the configuration, one or two tables are shown. They are the BHT and/or the BTAC. The data cache window shows a map of the data memory, with each entry representing a 4-byte word, as shown in Fig. 11.



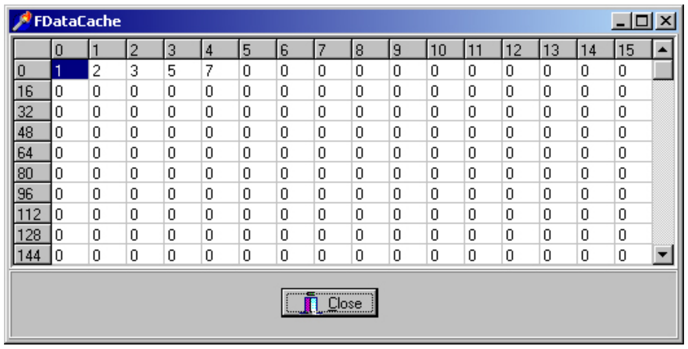
B H T

0	SNT	SNT	SNT	SNT
1	SNT	SNT	SNT	SNT
2	SNT	SNT	SNT	SNT
3	SNT	SNT	SNT	SNT
4	SNT	SNT	SNT	SNT
5	SNT	SNT	SNT	SNT
6	SNT	SNT	SNT	SNT
7	SNT	SNT	SNT	ST

B T A C

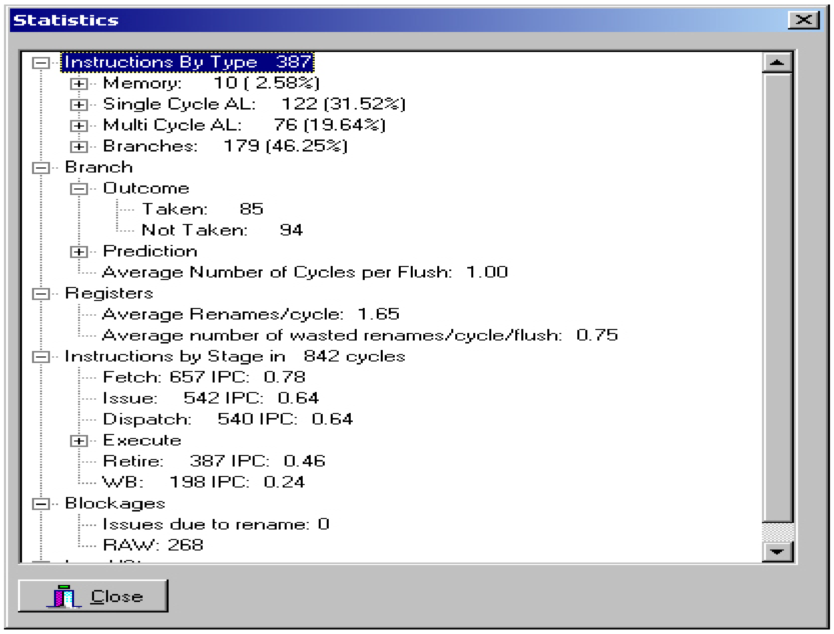
	Tag	B T A
20		
21		
22		
23	0	10
24		
25		
26		
27		
28		
29		
30		
31		

Figure 10: The Branch table's window



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	2	3	5	7	0	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
32	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
48	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
64	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
80	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
96	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
112	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
128	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
144	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 11: The Data cache window



```
Instructions By Type   387
    Memory:      10 ( 2.58%)
    Single Cycle AL:    122 (31.52%)
    Multi Cycle AL:     76 (19.64%)
    Branches:    179 (46.25%)
Branch
    Outcome
        Taken:     85
        Not Taken:     94
    Prediction
    Average Number of Cycles per Flush:  1.00
Registers
    Average Renames/cycle:  1.65
    Average number of wasted renames/cycle/flush:  0.75
Instructions by Stage in   842 cycles
    Fetch: 657 IPC:  0.78
    Issue:    542 IPC:  0.64
    Dispatch:    540 IPC:  0.64
    Execute
    Retire:    387 IPC:  0.46
    WB:    198 IPC:  0.24
Blockages
    Issues due to rename: 0
    RAW: 268
```

Figure 12: The Statistics window

The statistics window, shown in Fig. 12, gives a detailed statistics of the simulated code and configuration. The figures include the total number of executed instruction of each type, branch statistics and prediction accuracy measures, the flow of the instruction through each stage and both memory and register data dependencies. Some advanced measures are also included, like the average number of cycles required for flushing the processor and average number of register wasted when a miss-prediction occurred.

The instruction set of the simulator represents a subset of the standard modern instruction sets [4,6,10], and contains the instructions shown in table 1.

The simulator simulates a processor performing 32-bit integer operations with block diagram presented in Fig.13. The floating-point part is not considered in this project. Most of the current PostRISC features [4,6,10] can be simulated using the SuperSim, including out-of-order issue, register renaming, shelving, branch prediction etc. Supported memory addressing modes are displacement and indexed based [4]. While the same mnemonic is used for both modes, instruction processing is different depending on the mode. The memory is divided into instruction cache and 1024 locations of 32-bit words data cache. The memory is aligned on a word (4 bytes) boundary and all memory access instructions refer to a word address.

The maximum number of execution units is six (refer to Fig. 2). Instructions that take multiple clock cycles to execute, i.e. the 'mul' instruction, are executed in the multi-cycle, which is obligatory. Optionally there can be up to three single-cycle execution units for instructions like 'add', or 'sub' that take one clock cycle to execute, one load/store unit for handling memory access, and one branch unit dedicated for branch processing. When there is no available corresponding execution unit, the instructions are executed in the multi-cycle unit, which provides the functionality of all execution units. The number of execution units determines the dispatch rate so there are no restrictions about the instructions being dispatched. Issue rate can be set up to the dispatch rate [11].

Instruction	Semantics	Comment
ADD R1, R2, R3	Regs[1] = Regs[2] + Regs[3]	The third operand can be either register, or a constant
SUB R1, R2, R3	Regs[1] = Regs[2] - Regs[3]	
AND R1, R2, R3	Regs[1] = Regs[2] & Regs[3]	
OR R1, R2, R3	Regs[1] = Regs[2] | Regs[3]	
NOT R1, R2, RX	Regs[1] = ! Regs[2]	
SHL R1, R2, R3	Regs[1] = Regs[2] SHL Regs[3]	
SHR R1, R2, R3	Regs[1] = Regs[2] SHR Regs[3]	
MOD R1, R2, R3	Regs[1] = Regs[2] Modulo Regs[3]	
DIV R1, R2, R3	Regs[1] = Regs[2] / Regs[3]	
MUL R1, R2, R3	Regs[1] = Regs[2] * Regs[3]	
LOAD R1, R2, 200	Regs[1] = Mem[Regs[2] + 200]	Reads a word from memory
STORE R1, R2, 150	Mem[Regs[2] + 150] = Regs[1]	Writes a word in memory
BEQ R1, R2, 200	if (Regs[1]=Regs[2]) IP = IP+200	The third operand can be either register, or a constant
BNE R1, R2, R3	if (Regs[1]!=Regs[2]) IP = IP+Regs[3]	
BGT R1, R2, 200	if (Regs[1]>Regs[2]) IP = IP+200	
BLT R1, R2, R3	if (Regs[1]<Regs[2]) IP = IP+Regs[3]	
BGE R1, R2, 13	if (Regs[1]>=Regs[2]) IP = IP+13	
BLE R1, R2, R3	if (Regs[1]<=Regs[2]) IP = IP+Regs[3]	

Table 1: Instruction set

The use of RS is optional with the possibilities shown in Fig.3. When selected, there is a choice between central or dedicated RS. Out of order execution refers to whether instructions are issued out of order or dispatched out of order. When shelving is enabled instruction issuing is in order, while instruction dispatch is out of order. If shelving is disabled, the only possibility is out of order instruction issue. Fig.5 shows the possible options about out of order execution [14,15]. Branch processing options are shown in Fig.6. If branch processing is speculative, predictions about branch instructions can be: fixed "always not taken", static displacement based, or dynamic with optional use of BTAC, BHT or 2 bit global history register. In the latest case BTAC is used only for recent taken branches and the use of either BTAC or BHT is obligatory if dynamic prediction is selected [16]. Additionally, when BHT is used, global BHT can be activated and the initial state can be set.

4. Register renaming options of the SuperSim simulator

Our simulation tool uses ROB with up to 64 entries and can issue up to 6 instructions in one cycle. Since the visual orientation of the simulator and the fact that it simulates cycle-by-cycle instruction execution, we recommend using limited size programs. Using limited size programs will not provide misleading results, considering the fact that a very limited instruction window and issue rate is used. Several configuration parameters of the simulator determine the various implementations of register renaming. The simulator ISA has 32 general-purpose integer registers. The number of rename register can vary from 0 (no renaming) to maximum 256 renamed registers. Renamed registers are located in a separate rename register file. The simulator implements two different approaches to register mapping: indexed and associative. In the indexed mode each architecture register can be mapped to only one rename register, identified by its architectural number. This approach makes register access faster, but requires some software register optimization to achieve notable performance. The other approach uses associative register mapping, meaning that every architecture register can be mapped to many rename registers. This approach gives better performance and eliminates the need for special compiler register optimization, but the mapping hardware is more complex. Since the register renaming is made in order, the latest mapping of every architecture register is noted. This is done to insure that every following instruction in the program order gets the correct mappings of its source registers. The SuperSim uses late register freeing in the case of misspredicted branches. Late freeing has bigger pressure on the register file, but enables the precise interrupt handling [2,12]. The rename rate used is always equal to the issue rate of the selected configuration. Since it is a visual simulator, a visualization of both the register file and the rename register file is provided, as shown in

Fig. 9. Note that visualization is supported only if 32 or less rename registers are used. For each rename register, its logical number, value, valid bit and latest bit are shown.

5. Conclusion

The SuperSim simulator has a very precise implementation of register renaming. This implementation enables both educational and research applications of the simulator. The results obtained by the simulator are used in several papers [6,7] and are compatible with the results from other simulators and given by other authors[3].

6. References

1. Burger, D., Austin, T., (1997), The SimpleScalar Tool Set, Version 2, Technical report of the University of Wisconsin-Madison, Computer Science Department.
2. Farcas, K. et all, (1995) Register File Design Considerations in Dynamically Scheduled Processor, WRL Technical report 95/10, Paolo Alto, California.
3. Gonzalez, J. and Gonzalez, A., (1995) Identifying Contributing Factors to ILP, Univesitat Politecnica de Catalunya, Barcelona, Spain.
4. Gušev M. (1998), Contemporary Computer Systems, Medis, Skopje, Macedonia.
5. Gušev, M., Mišev, A. and Popovski, G. (1998) Simulation of Superscalar Processor, proc. of ITI'98, Pula, Croatia.
6. Gušev,M, Mišev.A and Popovski.G (1999), Memory Address Dependencies, ITI-99
7. Gušev,M, Mišev.A and Popovski.G (1999), The Impact Of Superscalar Techniques On Register Renaming, ITI-99, Pula, Croatia
8. Hennessy J.L., Patterson D.A. (1998), Computer Organization and Design: The Hardware Software Interface, sec. edition, Morgan Kaufmann Publishers.
9. Keller, R. M. (1975) Look-ahead processors, Computing Surveys, 7(4):177-195, 1975.
10. Patterson, D and Hennessy, J. (1996) Computer Architecture A Quantitative Approach, second edition, MKP, San Francisco, California.
11. Sima D. at al. (1997), Advanced Computer Architectures: A Design Space Approach, Addison Wesley Longman, Harlow, England.

12. Smith, J. and Pleszkun, A. (1988) Implementing Precise Interrupts in Pipelined Processors, IEEE Transactions on computers, vol. 37, no. 5, p.562-573.
13. Tjaden, G. S. and Flynn, M. J. (1970) Detection and parallel execution of independent instructions, IEEE-TC, vol. C-19, pp. 889--895, Oct. 1970.
14. Wall, D. (1993) Limits of Instruction-Level Parallelism, WRL Research report 93/6
15. Wall, D. (1994) Speculative Execution and Instruction-Level Parallelism, WRL Technical Note 94/42, Paolo Alto, California.
16. Yeh, T.-Y. and Patt, Y. N. (1992) Alternative implementations of two-level adaptive branch predictions. In 19th Annual International Symposium of Computer Architectures.