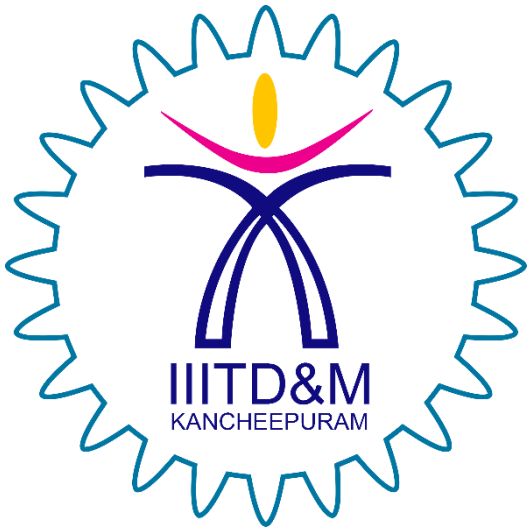


COMPUTER ARCHITECTURE PRACTICE

SCOREBOARD IMPLEMENTATION

Documentation

M.VINITHA

COE16B042

CHAPTER 1

GOAL:

The goal is the implementation of Scoreboard (Dynamic Scheduling Approach) to model In-order Issue, Out-of-order Execution and Out-of-order Commit. To see and understand how the Scoreboard takes care of all the data dependencies and structural hazards and works for set of instructions.

RELEVANT BACKGROUND INFORMATION:

ABOUT SCOREBOARD:

- Scoreboard is a centralized technique first used in the CDC 6600 computer which allows instructions to execute out of order when there are sufficient resources (hardware is available) and absence of data dependences.
- In a scoreboard, for each instruction the data dependencies are logged. The instructions are released only when the scoreboard determines that there are no conflicts with previously issued and incomplete instructions. In-order issue of instructions are ensured. An instruction that is unsafe to continue will be stalled. The scoreboard monitors the flow of executing instructions until all dependencies have been resolved before the stalled instruction is issued.
- Scoreboard does not take advantage of forwarding and the effects are reduced pipeline latency and benefits of forwarding.

Four main stages of Scoreboard:

Scoreboard replaced ID, EX, WB stages with 4 stages.

1. **Issue:** Decode the instructions and checks for structural hazards (ID1)
    - Instructions are issued in program order.
    - Checks for availability of functional unit- does not issue if structural hazard.
    - No WAW hazards- no active instruction that was earlier issued has the same destination register.
    - The instruction is stalled if a structural or WAW hazard occurs and no other instruction is issued until hazards are cleared.
    - If an instruction is issued, the scoreboard updates its internal data structure.
2. **Read Operands:** Wait for absence of data hazards, then read operands(ID2)
    - No RAW hazards/Checks for the availability of source operands- A source operand is available only if no other earlier issued active instruction is having that register as its destination (going to be written).
    - If the source operands are available, the functional unit proceeds to read the operands and begins execution (can be sent out of order).

3. **Execution:** operate on operands (EX)
    - Once the operands are read, the functional unit begins execution.
    - As soon as the result is ready, the scoreboard is notified that the execution is completed.
4. **Write Result:** finish execution (WB)
    - No WAR hazards- If an earlier issued instruction, still in the read operands stage has the destination register as one of its source register. Instruction is stalled if WAR hazard is found.
    - If no WAR hazards are found- the instruction write results.

Three parts to the scoreboard:

To control the execution of instructions, three status tables are maintained by the scoreboard.

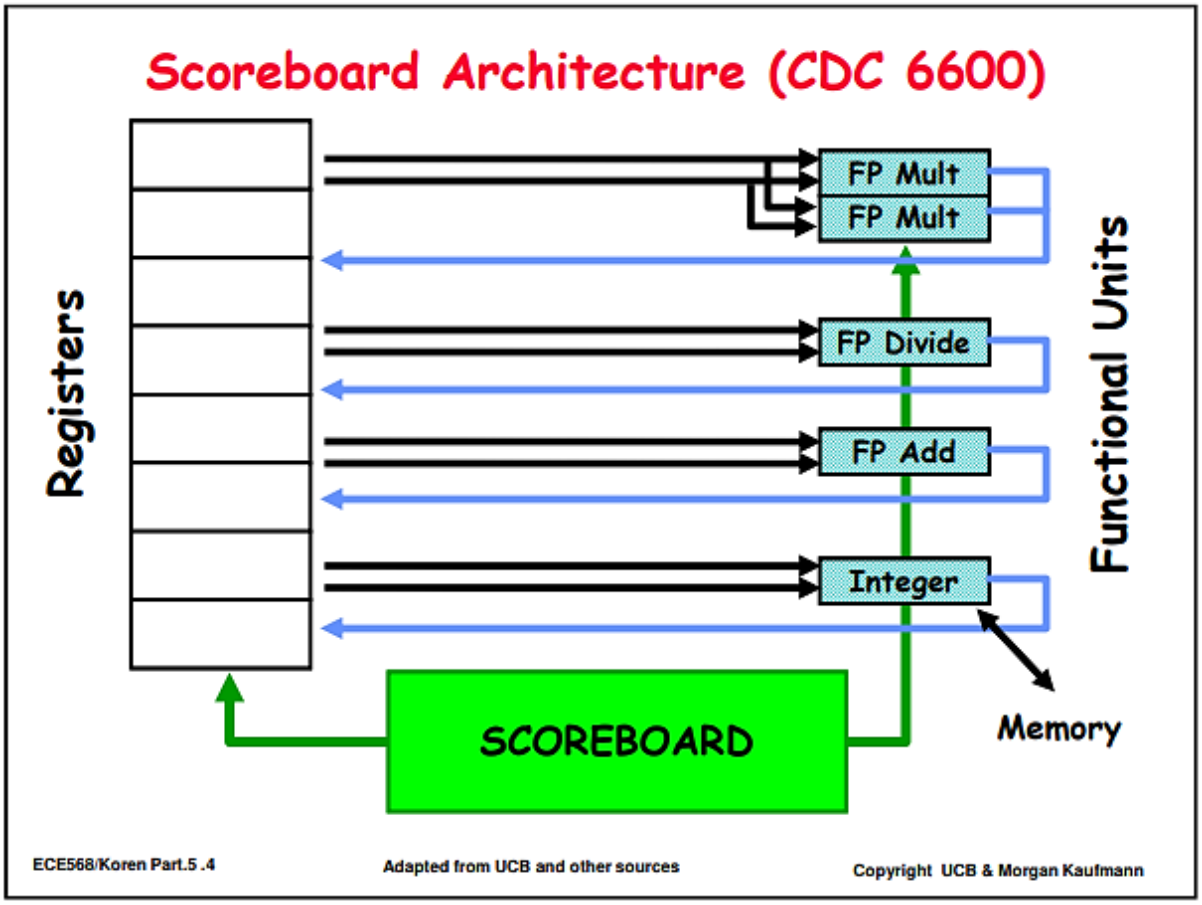
1) **Instruction Status:** Indicates for each instruction, which of the 4 stages the instruction is currently in.
2) **Functional Unit Status:** Indicates the state of each functional unit present. There are nine fields for each functional unit.
    - **Busy:** indicates whether the functional unit is busy or not.
    - **Op:** Operation to perform in the functional unit (eg: LD,MULD,ADDD).
    - **Fi:** Destination register.
    - **Fj,Fk:** Source registers.
    - **Qj,Qk:** Functional units producing source registers Fj, Fk.
    - **Rj,Rk:** Flags indicating when Fj, Fk are ready and not yet read. Set to No after operands are read.
3) **Register Status:** For each register, indicates which functional unit will write results into it. This field is set blank when there are no pending instructions that will write to that register.

Detailed Scoreboard Pipeline Control:

Op dest,S1,S2 = Op Fi,Fj,Fk

Instruction status	Wait until	Bookkeeping
Issue	Not busy (FU) and not Rg.Result(dest)	Busy(FU)← yes; Op(FU)← op; Fi(FU)← `Dest'; Fj(FU)← `S1'; Fk(FU)←`S2'; Qj←Rg.Result(`S1'); Qk←Rg.Result(`S2'); Rj← not Qj; Rk←not Qk; Rg.Result('dest')← FU;
Read operands	Rj and Rk	Rj← No; Rk← No
Execution complete	Functional unit done	
Write result	∀f((Fj(f)≠Fi(FU) or Rj(f)=No) & (Fk(f)≠Fi(FU) or Rk(f)=No))	∀f(if Qj(f)=FU then Rj(f)← Yes); ∀f(if Qk(f)=FU then Rk(f)← Yes); Rg.Result(Fi(FU))←0; Busy(FU)←No

The basic structure of a MIPS processor with a scoreboard:



TECHNOLOGY REQUIREMENTS:

The following code base is in C++. A minimum understanding of C++ is required to understand the code. For running the code, one needs to install g++ compiler.

To install the g++ compiler, type the following commands in terminal:

$ sudo apt-install g++

This will install g++ compiler.

$ sudo apt-get install build-essential

This will install the necessary C/C++ development libraries for your Ubuntu to create C/C++ programs.

$ gcc –version or gcc –v

To check the gcc or g++ version.

Code file structure:

- → fu.h
- → fu.cpp
- → scoreboard.cpp

To compile and execute the code, go to the directory where the files are stored directly from command line and then type the following commands:

$ g++ -o prog fu.cpp scoreboard.cpp

$ ./prog

The “-o prog” flag causes the executable to be named “prog” instead of the default “a.out”.

g++ will automatically include the fu.h file.

```
vinita@vinitha-hp-notebook:~/CA/SB$ g++ -o prog fu.cpp scoreboard.cpp
vinita@vinitha-hp-notebook:~/CA/SB$ ./prog
Enter instruction 1(Enter nil to stop reading instructions):
LD R6 0 R12
Enter instruction 2(Enter nil to stop reading instructions):
SD R2 0 R13
Enter instruction 3(Enter nil to stop reading instructions):
MULD R1 R2 R4
Enter instruction 4(Enter nil to stop reading instructions):
ADDD R6 R8 R2
Enter instruction 5(Enter nil to stop reading instructions):
nil
VCD info: dumpfile wallace_mul.vcd opened for output.
a=    2, b=    4, sum=         8
VCD info: dumpfile cla_adder1.vcd opened for output.
a=    8, b=    2, sum=    10

Instruction Status :
********************************************************************************
Issue   ReadOp  Execute Writeback
1       2       3       4
5       6       7       8
6       9       17      18
7       9       11      12
********************************************************************************

vinita@vinitha-hp-notebook:~/CA/SB$
```

The output displays the instructions and the instruction status table. For each instruction, at which clock cycles it was issued, read operands, completed execution and write result happened.

We make use of the Verilog HDL codes that we had done in the previous lab sessions to execute the functional units. The Verilog codes used here are:

- CLA adder
- Wallace multiplier
- Logical operations (AND, OR)

CHAPTER 2

Understanding the Code:

The scoreboard implementation has 3 major files:

The scoreboard.cpp, fu.cpp and fu.h files.

fu.h:

This is a header file that declares the following:

- Defines five **enumeration variables** representing 4 stages of instruction execution in scoreboard (sb_stage), Registers (reg), Operations of the instructions (operation), Functional Units (functional_unit)- 1 integer, 2 multipliers, 1 addition and 1 division, and Functional Unit Status fields (9 fields- BUSY, OPN, FI, FJ, FK, QJ, QK, RJ, RK).

```
4 enum sb_stage {ISS=1, RD, EXE, WB};
5 enum reg {R1=1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14};
6 enum operation {LD=1,SD,MULD,DIVD,ADDD,SUBD,OR,AND};
7 enum functional_unit {none, INT=1, MUL1, MUL2, ADD, DIV};
8 enum fields {BUSY=1, OPN, FI, FJ, FK, QJ, QK, RJ, RK};
```

- **Declaring** the Functional unit status table and the Register Status table and the **external** variables of the period, declaring the **structure** of the Instruction, The instruction-based structure declares the instruction set **class**.

```
22 extern int Func_Unit_Stat[6][10];
23 extern int Reg_Stat[15];
24 extern int Cyc_Count;
```

The extern keyword tells the compiler that a variable is declared in another source module (outside of the current scope).

```
13 struct COMMAND
14 {
15         int INS_NAME;
16         int LATENCY;
17         int DESTINATION;
18         int SOURCE1;
19         int SOURCE2;
20 };
```

The **structure COMMAND** of the Instruction contains the following data:

- INS_NAME: Operation performed by the instruction (eg: LD or ADDD)
- LATENCY: The number of cycles required to complete the execution of that operation in the functional unit.
- DESTINATION: The destination register of the instruction.
- SOURCE1: The source register 1 of the instruction.
- SOURCE2: The source register 2 of the instruction.

```
26 class COMMAND_STR
27 {
28 public:
29         COMMAND *c;
30
31         int command_stage[5];
32         int func_unit;
33         int lock;
34         int iss_cycle;
35         int rd_cycle;
36         int exe_cycle;
37         int wb_cycle;
38
39         COMMAND_STR() //Constructor
40         {
41                 this->c= new COMMAND;
42                 this->iss_cycle = 0;
43                 this->rd_cycle = 0;
44                 this->exe_cycle = 0;
45                 this->wb_cycle = 0;
46                 this->func_unit = 0;
47                 this->lock = 0;
48                 this->command_stage[0] = 0;
49                 this->command_stage[1] = 0;
50                 this->command_stage[2] = 0;
51                 this->command_stage[3] = 0;
52                 this->command_stage[4] = 0;
53         }
54
55         void stage_val(int m, int value);
56         void func_unit_val(functional_unit value);
57         void lock_val(int value);
58         void iss_val(int value);
59         void rd_val(int value);
60         void exe_val(int value);
61         void wb_val(int value);
62         int func_unit_get();
63         void Ins(char op_nam[], char reg_d[], char reg_src1[], char reg_src2[]);
64
65         ~COMMAND_STR() //Destructor
66         {
67                 delete c;
68         }
69 };
```

The instruction set **class COMMAND_STR** has the following **data members** and **member functions** in the public mode.

- command_stage[5]- The 4 stages of instruction execution in scoreboard: Issue, Read operands, Execution and Write results.
- func_unit- which functional unit is being used by the instruction.
- lock- initially set t 0, set to 1 just before entering execution stage and set to 0 after completing write result.
- iss_cycle- cycle number at which issue stage of that instruction is completed.
- rd_cycle- cycle number at which read operand stage of that instruction is completed.

- exe_cycle- cycle number at which execution stage of that instruction is completed.
- wb_cycle- cycle number at which writeback stage of that instruction is completed.

The constructor and the destructor has the same name as the class. The constructor is automatically invoked when a class object is created and the destructor is automatically invoked when the scope of the class object gets over. The **constructor** initialize values to object data members after storage is allocated to the object. The **destructor** destroys the class object.

fu.cpp :

This is a code file that defines the functions in the instruction set class.

```
7 void COMMAND_STR::Ins(char op_nam[], char reg_d[], char reg_src1[], char reg_src2[])
8 {
9         if (!strcmp("LD", op_nam))
10        {
11                c->INS_NAME = LD;
12                c->LATENCY = 1;
13        }
```

The member function Ins belonging to the COMMAND_STR class is used to find the operation performed by the instruction. If op_name matches with “LD”, then the INS_NAME is set to LD and LATENCY is set with the latency of that corresponding functional unit.

```
57        if (!strcmp("R1", reg_d))
58                c->DESTINATION = R1;
```

The reg_d is matched with anyone of the registers from R1-R14 and the DESTINATION is set accordingly with that register.

```
    if (!strcmp("R2", reg_src1))
            c->SOURCE1 = R2;
```

The reg_src1 is matched with anyone of the registers from R1-R14 and the SOURCE1 is set accordingly with that register.

```
    if (!strcmp("R7", reg_src2))
            c->SOURCE2 = R7;
```

The reg_src2 is matched with anyone of the registers from R1-R14 and the SOURCE2 is set accordingly with that register.

```
   if (!strcmp("0", reg_src1))
           c->SOURCE1 = 0;

   if (!strcmp("0", reg_src2))
           c->SOURCE2 = 0;
```

If the reg_src1 or reg_src2 is matched with 0, then the SOURCE1 or SOURCE2 is set to 0 accordingly. 0 is given as source register in case of LD or SD instructions where only 1 source register is needed. So the other register is set to 0.

```
void COMMAND_STR::func_unit_val(functional_unit value)
{
        this->func_unit = value;
}
void COMMAND_STR::stage_val(int m, int value)
{
        this->command_stage[m] = value;
}
void COMMAND_STR::lock_val(int value)
{
        this->lock = value;
}
void COMMAND_STR::iss_val(int value)
{
        this->iss_cycle = value;
}
void COMMAND_STR::rd_val(int value)
{
        this->rd_cycle = value;
}
void COMMAND_STR::exe_val(int value)
{
        this->exe_cycle = value;
}
void COMMAND_STR::wb_val(int value)
{
        this->wb_cycle = value;
}
int COMMAND_STR::func_unit_get()
{
        return this->func_unit;
}
```

The remaining data members of the COMMAND_STR class are assigned values using the corresponding member functions.

scoreboard.cpp:

This is a code file that gives a concrete implementation of the Scoreboard algorithm.

- First, initialize the Functional Unit status and Register Status tables, then get the set of instructions from the user, the perform dynamic scheduling approach (Scoreboard) and finally output the Instruction status table.

```
void Begin_Stat()       //Initialising the Functional Unit Status and Register Status with 0
{
        for (int m=0; m<6; m++)
        {
                for (int n=0; n<10; n++)
                {
                        Func_Unit_Stat[m][n] = 0;
                }
        }

        for (int k=0; k<15; k++)
                Reg_Stat[k] = 0;
}
```

On invoking the **Begin_Stat()** function in the int main() function, the Functional Unit Status and the Register Status are **initially assigned with 0**. That is all the 9 fields for the 5 functional units and the registers from R1-R14 are initialized to 0 in the status tables.

```
int Reading_Instructions()      //Reading Instructions from the user and stops reading when "nil" is encountered
{
        int num = 0;
        char opname[10] = "", regd[5] = "", reg1[5] = "", reg2[5] = "";
        cout << "Enter instruction "<<num+1<<"(Enter nil to stop reading instructions):" << endl;
        cin >> opname;
        while (strcmp(opname, "nil") != 0)
        {
                cin >> regd >> reg1 >> reg2;
                com[++num].Ins(opname, regd, reg1, reg2);
                cout << "Enter instruction "<<num+1<<"(Enter nil to stop reading instructions):" << endl;
                cin >> opname;
        }
        return num;
}
```

To **read a set of instructions from the user** the **Reading_Instructions()** function is invoked in the int main() function. The variable **num** has the count of the number of instructions given by the user. The opname (operation name), regd (destinantion register), reg1 (source register 1) and reg2 (source register 2) is read individually for each instruction and the Ins() member function is called for each instruction to assign the corresponding values.

- **Parallelism in Scoreboard:**
  We have adopted a time-driven approach, that is, the number of cycles will be added 1. For every clock cycle, all instructions in the instruction set are traversed once during that cycle, to check at which stage each instruction is at and whether it can move on to the next stage.

```
void scoreboard(int num, int flag = 0)  //Implementation of Scoreboard for the Instructions
{
  int ctrl = 1;
    while (ctrl)
    {
        for (int m=1; m<=num; m++)
        {
            if (com[m].command_stage[ISS] == 0)
            {
                Issue_Stage(m, num);
            }
            else
            {
                if (com[m].command_stage[RD] == 0)
                {
                    RdOp_Stage(m);
                }
                else
                {
                    if (com[m].command_stage[EXE] == 0 && Cyc_Count - com[m].rd_cycle == com[m].c->LATENCY)
                    {
                        Execution_Stage(m);
                    }
                    else
                    {
                        if (com[m].command_stage[EXE] != 0  && com[m].command_stage[WB] == 0)
                        {
                            WriteBack_Stage(m);
                        }
                    }
                }
            }
```

At each clock cycle, we go through every instruction and check whether each instruction can be issued or read operands or executed or write results. In this way we ensure parallelism.

```
    int d;
    for (d = 1; d <= num; d++)
    {
        if (com[d].command_stage[WB] == 0)
        {
            ctrl = 1;
            break;
        }
    }
```

Checks whether all the instructions in the set of instructions given by the user have been written back (write result stage is completed). If command_stage[WB]==0 means that write result stage is not completed. If completed then the program is finished.

```
        if (d > num)
            ctrl = 0;

        if (flag != 0)
        {
            if (Cyc_Count == flag)
                return;
        }
        Cyc_Count++;
    }
}
```

If all the instructions have not yet been written back, then the algorithm has to continue. So, the cycle count is incremented by 1.

- To determine at which stage each instruction is at, we need four variables for each instruction in the instruction set class to represent the number of cycles at the beginning of the four stages. The value is 0 if that stage has not been reached yet.
- To determine whether you can move on to the next stage is done through four functions (Issue_Stage, RdOp_Stage, Execution_Stage, WriteBack_Stage).
- The first function determines whether the instruction can be issued or not.

```
void Issue_Stage(int n, int num)
{
        int t = n-1;
        switch (com[n].c->INS_NAME)
        {
        case LD:
                if ((Func_Unit_Stat[INT][BUSY] == NO) && (com[n].command_stage[ISS] == 0))
                {
                        if ((t > 0 && (com[t].command_stage[ISS] != 0) && (Cyc_Count > com[t].iss_cycle) && (Cyc_Count != com[t].wb_cycle))
|| t == 0)
                        {
                                com[n].func_unit_val(INT);

                                for (int k = 1; k <= num; k++)
                                {
                                        if (com[k].lock != 0)
                                        {
                                                if (com[k].c->DESTINATION == com[n].c->DESTINATION)
                                                        return;
                                        }
                                }

                                Change_Stat(INT, com[n].c, n);
                                break;
                        }
                        else
                                return;
                }
                else
                        return;
```

- . It checks for 3 criterions:
  - Inorder issue of instructions (if the previous instruction in program order has not been issued then the current instruction cannot be issued).
  - Functional Unit available (if functional unit is not available i.e there is a structural hazard, then the instruction is stalled).
  - No WAW hazards (no active instruction that was earlier issued has the same destination register as the current instruction).

```
void Change_Stat(int func_unit, COMMAND *c, int n)      //Function that alters the Functional Unit Status
{
        Func_Unit_Stat[func_unit][BUSY] = YES;
        Func_Unit_Stat[func_unit][OPN] = c->INS_NAME;
        Func_Unit_Stat[func_unit][FI] = c->DESTINATION;
        Func_Unit_Stat[func_unit][FJ] = c->SOURCE1;
        Func_Unit_Stat[func_unit][FK] = c->SOURCE2;

        if (Func_Unit_Stat[func_unit][FJ] > 10 || Func_Unit_Stat[func_unit][FJ] == 0)
                Func_Unit_Stat[func_unit][RJ] = YES;
        if (Func_Unit_Stat[func_unit][FK] > 10 || Func_Unit_Stat[func_unit][FK] == 0)
                Func_Unit_Stat[func_unit][RK] = YES;

        int ctrl = 1;
        for (int k=1; k<n; k++)
        {
                if (Func_Unit_Stat[com[k].func_unit_get()][BUSY] == 1)
                {
                        if (com[k].c->DESTINATION == c->SOURCE1)
                        {
                                ctrl = 0;
                                Func_Unit_Stat[func_unit][QJ] = com[k].func_unit_get();
                                break;
                        }
                }
        }
        if (ctrl)
                Func_Unit_Stat[func_unit][RJ] = YES;
```

```
        ctrl = 1;
        for (int k=1; k<n; k++)
        {
                if (Func_Unit_Stat[com[k].func_unit_get()][BUSY] == 1)
                {
                        if (com[k].c->DESTINATION == c->SOURCE2)
                        {
                                ctrl = 0;
                                Func_Unit_Stat[func_unit][QK] = com[k].func_unit_get();
                                break;
                        }
                }
        }
        if (ctrl)
                Func_Unit_Stat[func_unit][RK] = YES;

        Reg_Stat[c->DESTINATION] = func_unit;
}
```

- If an instruction is issued, the scoreboard updates its internal data structure by invoking Change_Stat() function which alters Functional Unit Status table.

```
401         com[n].iss_val(Cyc_Count);
402         com[n].stage_val(ISS, Cyc_Count);
403 }
```

Cycle number is assigned to iss_val. It is the cycle number at which an instruction is issued.

- The second function determines whether to read the operands or not.

```
void RdOp_Stage(int n)
{
        int func_unit = com[n].func_unit_get();
        int tRJ = Func_Unit_Stat[func_unit][RJ];
        int tRK = Func_Unit_Stat[func_unit][RK];

        for (int k=1; k<=n; k++)
        {
                if (com[k].lock != 0)
                {
                        if ((com[k].c->DESTINATION == com[n].c->SOURCE1) || (com[k].c->DESTINATION == com[n].c->SOURCE2) || tRJ == 0 || tRK
== 0)
                        {
                                return;
                        }
                }
                else
                {
                        if ((com[k].c->DESTINATION == com[n].c->SOURCE1) || (com[k].c->DESTINATION == com[n].c->SOURCE2))
                        {
                                if (com[k].c->DESTINATION == com[n].c->SOURCE1)
                                {
                                        Func_Unit_Stat[func_unit][QJ] = 0;
                                        Func_Unit_Stat[func_unit][RJ] = YES;
                                }
                                if (com[k].c->DESTINATION == com[n].c->SOURCE2)
                                {
                                        Func_Unit_Stat[func_unit][QK] = 0;
                                        Func_Unit_Stat[func_unit][RK] = YES;
                                }
                        }
                }
```

- To check whether source registers are available and instruction can read operands (A source operand is available only if no other earlier issued active instruction is having that register as its destination (going to be written).
- This prevents RAW hazards
- If the source operands are available, the functional unit proceeds to read the operands and begins execution (can be sent out of order).

```
        com[n].rd_val(Cyc_Count);
        com[n].stage_val(RD, Cyc_Count);
        com[n].lock_val(1);
}
```

Cycle number is assigned to rd_val. It is the cycle number at which an instruction reads operands. The lock value is assigned 1 indicating that it is ready for execution.

- The third function proceeds the instruction for execution in the functional unit if the operands are read.

```
void Execution_Stage(int n)
{
//Instruction proceeds for execution in functional unit if operands are read
        com[n].exe_val(Cyc_Count);
        com[n].stage_val(EXE, Cyc_Count);
}
```

Cycle number is assigned to exe_val. It is the cycle number at which an instruction is executed.

- The fourth function determines whether the instruction can be written back or not.

```
void WriteBack_Stage(int n)
{
//Checks whether instruction is ready for writeback or not
//Ensures no WAR hazard occurs
        for (int k=1; k<=n; k++)
        {
                if (com[k].lock != 0)
                {
                        if ((com[k].c->SOURCE1 == com[n].c->DESTINATION) || (com[k].c->SOURCE2 == com[n].c->DESTINATION))
                                return;
                }
        }

        com[n].wb_val(Cyc_Count);
        com[n].stage_val(WB, Cyc_Count);
}
```

- Checks whether instruction is ready for writeback or not.
- Ensures no WAR hazard occurs (If an earlier issued instruction, still in the read operands stage has the destination register as one of its source register.) Instruction is stalled if WAR hazard is found.
- If no WAR hazard is found, then the instruction writes result.

```
if (com[m].command_stage[WB] != 0 && com[m].lock == 1)
{
        int fu_temp = com[m].func_unit_get();
        Func_Unit_Stat[fu_temp][BUSY] = NO;
        com[m].lock_val(0);
        com[m].func_unit_val(none);
}
```

Once write results of an instruction is completed, the lock is changed to 0 again and the functional unit is set to NO(t) BUSY.

- Finally, if all the instructions are written back, then the Instruction Status is displayed and the program is completed.

```
void display_status(int num)            //Display the final Instruction Status
{
        cout << '\n';
        cout << "Instruction Status :"<< endl;
        cout << "*********************************************************************" << endl;
        cout << "Issue   ReadOp  Execute Writeback\n";
        for (int n=1; n<=num; n++)
        {
                for (int k=1; k<5; k++)
                {
                        if (com[n].command_stage[k] == 0)
                                cout << " " << '\t';
                        else
                                cout << com[n].command_stage[k] << '\t';
                }
                cout << '\n';
        }
        cout << "*********************************************************************" << endl << endl;
}
```

For each instruction, it is displayed at which clock cycle it was issued, read operands, completed execution and write result happened.

Output:

```
vinita@vinitha-hp-notebook:~/CA/SB$ g++ -o prog fu.cpp scoreboard.cpp
vinita@vinitha-hp-notebook:~/CA/SB$ ./prog
Enter instruction 1(Enter nil to stop reading instructions):
LD R6 0 R12
Enter instruction 2(Enter nil to stop reading instructions):
SD R2 0 R13
Enter instruction 3(Enter nil to stop reading instructions):
MULD R1 R2 R4
Enter instruction 4(Enter nil to stop reading instructions):
ADDD R6 R8 R2
Enter instruction 5(Enter nil to stop reading instructions):
nil
VCD info: dumpfile wallace_mul.vcd opened for output.
a=    2, b=    4, sum=         8
VCD info: dumpfile cla_adder1.vcd opened for output.
a=    8, b=    2, sum=    10

Instruction Status :
********************************************************************************
Issue   ReadOp  Execute Writeback
1       2       3       4
5       6       7       8
6       9       17      18
7       9       11      12
********************************************************************************

vinita@vinitha-hp-notebook:~/CA/SB$
```

Verilog Integration:

```
ofstream file;                          //Integration of verilog code (CLA)
file.open ("testtb.v");
file<<"module top;";
file<<"\nreg [15:0]a;";
file<<"\nreg [15:0]b;";
file<<"\nwire [16:0]s;";
file<<"\nAdd cla_0(a,b,s);";
file<<"\ninitial";
file<<"\nbegin";
file<<"\na=8;";
file<<"\nb=2;";
file<<"\nend";
file<<"\ninitial";
file<<"\nbegin";
file<<"\n$monitor(\"a=%d, b=%d, sum=%d\",a,b,s);";
file<<"\n$dumpfile(\"cla_adder1.vcd\");";
file<<"\n$dumpvars(0,top);";
file<<"\nend";
file<<"\nendmodule";
file.close();
system("iverilog -o cla KGP.v testtb.v");
system("vvp cla");
        break;
}
```

Here we are using the functional units made in Verilog HDL during our previous lab sessions.

The Verilog codes used here are:

- CLA adder
- Wallace multiplier
- Logical operations (AND, OR)
-

A verilog file is created and the input is given. The input is sent to the Verilog testbench file and the output is displayed. The latency for each functional unit is considered based on the delay of that functional unit as per cadence.

The library function system(const char *command) passes the command name or program name specified by command to the host environment to be executed by the command processor and returns after the command has been completed.

CHAPTER 3

Summary:

In **Dynamic Scheduling** the hardware rearranges the instruction execution to reduce stalls.

Advantages of Dynamic Scheduling:

● It enables handling cases when dependencies are unknown at compile time (e.g., because they may involve a memory reference).

● It simplifies the compiler.

● It allows code that was compiled with one pipeline in mind to run efficiently on a different pipeline.

However, these advantages are gained at a cost of significant increase in hardware complexity.

Limitations of 6600 Scoreboard:

- No forwarding hardware.
- Limited to instructions in basic block (small window).
- Small number of functional units (structural hazards), especially integer/load store units.
- Does not issue on structural hazards.
- Stalls for WAR hazards.
- Stalls WAW hazards.

In this Implementation of Scoreboard we have learnt and unserstood what dynamic scheduling approach is and how to achieve it. We have also understood how scoreboard reduces the pipeline latency. We have implemented Scoreboard for MIPS 5 stage pipelined architecture.