Homework Assignment 3

Wednesday, October 9, 2019 3:01 PM

Name : VIKAS VEERABATHINI
NU ID : 001302155

Part A:

For any two current embedded processors (e.g., ARM7, ARM9, Intel 8051, TI MSP430, MIPS32), provide the details of the microarchitecture of the pipeline (e.g., the width of the data path, the number of pipeline stages, number of instructions issued per cycles, the number of integer units, the branch resolution unit, the maximum number of instructions in flight, etc.). You might not find every one of these details, but provide at least 3 different pipeline characteristics that you can compare.

Name of processor	Width of the data path	Number of pipeline stages	Number of instructions issued per cycle	Number of integer units	Branch resolution unit	Maximum number of instructions in flight
Arm 7	32-bit (only single bus) .	3	2 integer instructions (assuming it to be a single unit processor) + 1 floating point instruction (assuming it to be a single unit processor) = 3 . If we are assuming it to have only one unit , then it would be 1 instruction per cycle	2	It uses branch prediction algorithms or say dynamic branch prediction for branch prediction .	9 = 3*3 (A pipeline on a single unit can process 3 instructions , since we have 3 such units we conclude that we can process 3*3 = 9 9 instructions in flight , assuming it to be a single core system) . If we are assuming it to have only one unit , then it would be 3 instructions
Arm 9	32-bit (1 bus for data and other for instructions)	5	2 integer instructions (assuming it to be a single unit processor) + 1 floating point instruction (assuming it to be a single unit processor) . It does have multiple other units like SIMD for vectorization which we aint accounting here though . If we are assuming it to have only one unit , then it would be 1 instruction per cycle	2	2-level global history branch predictor . (Dynamic Branch Prediction)	15 = 3*5 (A pipeline on a single unit can process 5 instructions , since we have 3 such units we conclude that we can process 3*5 = 15 such instructions in flight , assuming it be a single core system) . If we are assuming it to have only one unit , then it would be 5 instructions

Part B .

Read the paper on superscalar vs. superpipelining by Jouppi and Wall, and the paper on the optimal pipeline depth by Hartstein and Puzak. Then using these 2 papers, discuss the tradeoffs between these two different microarchitectural approaches. You should develop an argument for using one of these two microarchitectures based on your analysis.

Both the papers discuss on what should be a optimum pipeline depth for a processor but with different approaches .
Jouppi and wall , start by presenting various types of machines (viz., Base , Superscalar , Underpipelined , Superscalar Superpipelined)
Base : Has one pipeline stage with operation latency of 1 and only one processing unit . Only one instruction can be issued per cycle
Superscalar of 'n' degree : Has multiple units of processing unit , thereby giving room to multiple instructions being executed per cycle based on **'n'**
Superpipelined of 'n' degree : single pipeline has 'n' stages \ depth , although only instruction can be issued per cycle, but in ideal case . We can see that n instructions being executed in a single cycle .
Underpipelined : It takes more than one cycle to execute 1 instruction in an ideal stage , thereby making this machine obsolete . Few examples are RISC-II chips .
In a direct comparison of superscalar and superpipelined , superscalar machines require a single clock cycle to complete 'n' instructions while superpipelined machine would require a single pipeline cycle which falls 2/3 shorter . Thus , superpipelined machine yields a better performance than that of a superscalar machine when it comes to operation latency in seconds .
But when it comes to machine being started superpipeline machine takes a longer time to complete 'n' instructions than that of superscalar owing to it's start transient behavior .
Thus , Inorder to overcome above constraints we usually use a superscalar superpipeline machine which would eventually lead to n*n instructions being executed .
As per various code optimizations performed it looks like we can achieve only 4-degree of instruction level parallelism at highest level , thus we need to chose a superscalar(n) superpipelined(m) machine , where n*m = 4 .
Apart from above factors while profiling a machine it is necessary to account for cache misses and hits too . If the instructions are way to large , then we may end up in cache thrashing while pulling those instructions into instructions cache . In a nutshell , jouppi and wall summarize that whether it pipeline depth or number of functional units in their respective machines , their 'degree' is shunted by number of instructions that can be run in parallel in a worst case which turns out to be 4 as per their simulation results

Harstein and puzak , try to study only superpipeline machines in detail and approach the problem of required stages for a superpipeline machines considering factors like power and performance .
Metric used by them are BIPS / W , BIPS , BIPS^2 / W , BIPS ^ 3 / W . BIPS \W : Billions instructions per second when a single watt of power is consumed .
It goes ahead by giving out an equation which takes into account dynamic power (Pd) , leakage power (Pl) , fc (clock gating frequency) , fcg (non clock gating frequency) , pipeline stages (p) , Number of latches . Below is the equation .

$$P_T = (f_{cg} f_s P_d + P_l) N_L p^{\eta} ,$$

Inorder to fetch practical results , SPECint benchmark is run on the system (by having clock gated as well as non clock gated) . Then it goes ahead , by plotting the metric against number of pipeline stages .

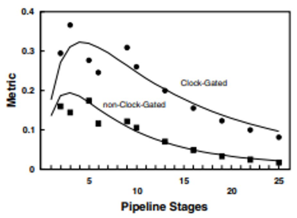


Fig. 4b shows a comparison of theory and simulation for a SPECint workload, both with and without clock-gating.

Above plot conveys that for non-clock gated machine , metric is lower as it consumes more power as compared to that of clock-gated machine . And looks like we gain an optimum pipeline depth of approximately 4 in this case .

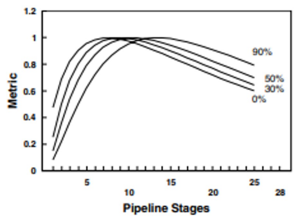


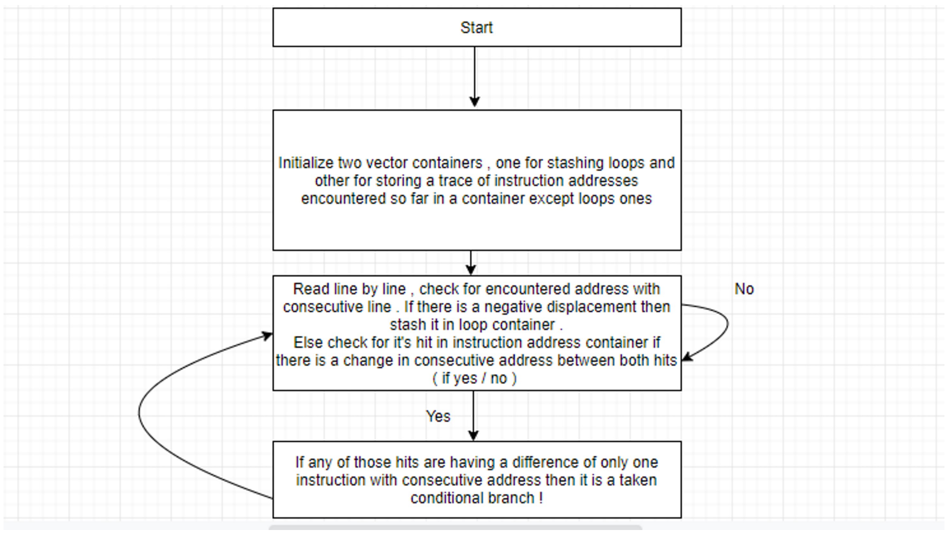
Fig. 8 shows how the optimum pipeline depth varies with increasing leakage power.

It also goes ahead to study the effect of change in leakage power (in percent) of total power against pipeline depth . Increase in leakage power (due to increase in latches) drives the peak to right , indicating more pipeline stages required while it is reverse in case of dynamic power .

Concluding : For an embedded system with more importance for power , I would go ahead with hartstein and puzak approach for using a microarchitecture . But for a general PC , we should go ahead with approach by jouppi and wall , which is driven by how a compiler Is designed to give an effective parallelism with more inclination towards speed rather than power .

Part C : We are providing you with an instruction trace named itrace.out.gz (the file has been compressed with gzip, so you will need to decompress it before using it). The trace simply lists the 64-bit instruction address for each instruction executed (addresses are in hexadecimal). You are working with an x86-64 instruction trace, so you will need to figure out which addresses are taken conditional branch addresses. There are multiple ways to accomplish this. You will be writing a simulator to simulate two simple conditional branch predictors. The first predictor will have 32, 1-bit predictor, saving the action of the last branch to predict the outcome of the next branch. The second predictor will use 16, 2-bit counters, to predict branches. Assume initially that the 2-bit counters are initialized to be weakly not-taken. Write a simple simulator that models each of the 2 predictors and processes the instruction trace provided. Submit both the code for your branch predictor simulator (on Turnitin) and report on the number of buffer misses (first time taken branches), and the number of correct and incorrect predictions for this address for this address trace for each predictor. Add a discussion explaining why one predictor does better than the other. As an added step, experiment with other prediction algorithms (besides a 2-bit counter).
You only have 8 entries in total for these alternative algorithms. The best performing algorithm will receive a special prize.

Algorithm to find out a taken conditional branch from an instruction address trace .


Start
Initialize two vector containers , one for stashing loops and other for storing a trace of instruction addresses encountered so far in a container except loops ones
Read line by line , check for encountered address with consecutive line . If there is a negative displacement then stash it in loop container .
Else check for it's hit in instruction address container if there is a change in consecutive address between both hits (if yes / no)
No
Yes
If any of those hits are having a difference of only one instruction with consecutive address then it is a taken conditional branch !

1-bit and 2-Bit Branch predictor counters have a similar design as in text , with APIs representing **BranchPredictTwoBit** and **BranchPredictorOneBit** their C-implementation . For ease of running and representing data We are running for entire program , but data represents per 100000 lines of instructions against the algorithm .

Program Output :

```
Number of Taken conditional branches is : 9916
Number of Non Taken conditional branches is : 89997
Number of Conditional Branches is : 99913
```

```
1-bit branch predictor -  Number of Mispredictions : 50799 NumCorrectPredictions : 49114
2-bit branch predictor - Number of Mispredictions : 5079 NumCorrectPredictions : 94834
```

It looks like Program is very dense with branches , with 9916 being taken ones and other being non-taken ones .
From Program Output , it looks like Prediction rate for a 1-bit branch predictor is **49.15%** .
For a 2-bit branch predictor it is around 94.91% .

2-bit branch predictor has a better performance than 1-bit branch predictor owing to Hysteresis and having lesser interference from other branches since we would be using 16 2-bit counters .
Reducing interference further would increase the prediction rate .

Alternative Algorithm :

Two factor's that majorly influence a branch's direction are operations and other branches in it's near vicinity .
For any branch , a nearby vicinity would be nonetheless the page in which branch instruction is situated and it's Nearby pages , especially previous and next page .
So I came up with an algorithm which takes into account , number of previously taken and non-taken branches in the current Page scaled with a weight(w1) , number of previously taken and non-taken branches in the neighboring Pages scaled with weights(w2 , w3) .

Pseudo-Code Steps :

```
1 . Fetch Conditional Branch Address , Calculate the page of the same . Check for it's hit in Page table .
2 . If there is no hit , then predict it to be not-taken and make a new sorted entry of the same in Page Table .
3 . If there is a hit , then
    calculate deciding factor (y) = w1*(p1.takenbranches - p1.nontakenbranches) + w2*(p1n.takenbranches - p1n.nontakenbranches) +
                                    w3*(p1n2.takenbranches - p1n2.nontakenbranches)
    Compare , 'y' with a threshold if it exceeds then predict branch to be taken . Otherwise , predict branch to be
    non-taken .
4 . If it is an incorrect prediction , then for taken misprediction increase the threshold while for a non taken misprediction
   decrease the threshold so that we can create a minor hysteresis in case if there are high mispredictions .
5 . Continue to step 1 .
```

Sample Output :

```
Special Algorithm branch predictor - Number of Mispredictions : 3006 NumCorrectPredictions : 96907
```

It gives a prediction rate of 96% approx .
One more important feature about this algorithm is :
We can modify or train w1 , w2 , w3 and Threshold to get improved results . At a correct instance it eventually gives A prediction rate upto 98% .

Few key things about algorithm to scan taken conditional branches from Instruction Trace :
1 . I have restricted the main loop to fetch only 100000 lines as keep it open to run for entire trace takes couple of hours On COE linux systems . But can be toggled in code easily
2 . Filename to be read in input iterator is by default "itrace.out" , but can be modified in source code if any other Instruction trace Needs to be toggled .
3 . Please use **-O3 and -std=c++11** flag for compilation to import necessary libraries

Part D : To build a better understanding of pipelining concepts, complete Problems C.1 (parts ag), C.2 (parts a-b) and C.7 (parts a-b) in the textbook in Appendix C. Make sure to state any assumptions made when answering these problems.

A . < Question from textbook >

(WAW : write after write (output dependency) , WAR : write after read (anti-depedency) , RAW : read after write (true dependency))

WAW , RAW dependency on x1 register between (ld and addi) .

```
ld x1,0(x2);
addi x1,x1,1;
```

WAW and WAR dependency on x1 register between (sd and addi)

```
addi x1,x1,1
sd x1,0,(x2)
```

WAR dependency (sd and addi)

```
sd x1,0,(x2)
addi x2,x2,4
```

RAW dependency between (sub and addi) and (sub and bnez)

```
addi x2,x2,4
sub x4,x3,x2
```

B . < Question from textbook >

Pipeline chart with register forwarding applicable only in WB stage is as follows :

Ld x1,0(x2)	IF	ID	EX	MEM	WB										
Addi x1,x1,1		IF	ID	Stall	EX	MEM	WB								
Sd x1,0,(x2)			IF	ID	Stall	Stall	EX	MEM	WB						
ADDi x2,x2,4				IF	ID	Stall	Stall	Stall	EX	MEM	WB				
SUB x4,x3,x2					IF	ID	Stall	Stall	Stall	Stall	EX	MEM	WB		
Bnez x4,loop						IF	ID	Stall	Stall	Stall	Stall	Stall	EX	MEM	WB

From above one loop takes about 15 cycles , since we are waiting for the branch to flush the pipeline and go back to instruction 1 .
Since there are around 99 such loops with last loop taking around (500 instructions) ,
we would take around 99*15 = **1,485** cycles .
(initial value of x3 = x2 + 396 , assuming x2 was zero , we need to wait for 396 to become zero with it being deducted by 4 each time which comes as 100)

C . < Question from textbook >

Ld x1,0(x2)	IF	ID	EX	MEM	WB				
Addi x1,x1,1		IF	ID	EX	MEM	WB			
Sd x1,0,(x2)			IF	ID	EX	MEM	WB		
ADDi x2,x2,4				IF	ID	EX	MEM	WB	
SUB x4,x3,x2					IF	ID	EX	MEM	WB
Bnez x4,loop						IF	ID	EX	MEM	WB

From above one loop takes about 10 cycles , since we are waiting for the branch to be in non taken state which we are assuming to exit from loop .
Once resolved it would then lookup and come back to instruction 1 again , but eventually it would take the entire pipeline to flush to resolve the decision .
Since there are around 99 such iterations, it would be 10*99 = **990** cycles .

D . < Question from textbook >

Ld x1,0(x2)	IF	ID	EX	MEM	WB						
Addi x1,x1,1		IF	ID	EX	MEM	WB					
Sd x1,0,(x2)			IF	ID	EX	MEM	WB				
ADDi x2,x2,4				IF	ID	EX	MEM	WB			
SUB x4,x3,x2					IF	ID	EX	MEM	WB		
Bnez x4,loop						IF	ID	EX	MEM	WB	
Ld x1,0(x2)							IF	ID	EX	MEM	WB

From above one loop takes about 6 cycles , since we are predicting that branch would be taken which we assume that it would go stay in loop .
This would match with resolution for around 100 iterations and at last iteration it would mispredict leading to revert of 4 cycles
Thus , we would take around 99*6 + 4 = **598** cycles .

E . < Question from textbook >

Ld x1,0(x2)	IF1	IF1	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2										
Addi x1,x1,1		IF1	IF2	ID1	ID2	(1-cycle delay)	EX1	EX2	MEM1	MEM2	WB1	WB2								
Sd x1,0,(x2)			IF1	IF2	ID1	ID2	(stall)	(1-cycle delay)	EX1	EX2	MEM1	MEM2	WB1	WB2						
ADDi x2,x2,4				IF1	IF2	ID1	ID2	Stall	Stall	1-cycle delay	EX1	EX2	MEM1	MEM2	WB1	WB2				
SUB x4,x3,x2					IF1	IF2	ID1	ID2	Stall	Stall	Stall	1-cycle delay	EX1	EX2	MEM1	MEM2	WB1	WB2		
Bnez x4,loop						IF1	IF2	ID1	ID2	Stall	Stall	Stall	Stall	1-cycle delay	EX1	EX2	MEM1	MEM2	WB1	WB2
Ld x1,0(x2)							IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2				

As we can see above from the pipeline diagram , from the fetch of first instruction of first iteration to that of second iteration it takes around 6 cycles and
For the pipeline to flush at the end it would take around 15 more cycles to resolve the prediction decision after a misprediction .
Thus it would be 6*99+15 = **609 cycles** for the loop to complete .

F . < Question from textbook >

Below is a skeleton of a pipeline of a 5-stage with pipeline registers (Pr) .

IF - Pr - ID - Pr - EX - Pr - MEM - Pr - WB

Since longest stage is taking around 0.8ns lets assume that each stage takes that amount of time .
Thus we would have 0.8+0.1+0.8+0.1+0.8+0.1+0.8+0.1+0.8 = **4.4** ns for an instruction to complete .

Since a 10-stage pipeline would eventually split each stage in two (Like IF1 . IF2 . ID1 . ID2 . EX1 . EX2 . MEM1 . MEM2 . WB1 . WB2)
We are assuming we need to put a pipeline register between each sub stage too for register forwarding it would be something like below
And each stage would eventually take 0.8/2 = 0.4 ns since splitting would divide the clock too

IF1 - Pr - IF2 - Pr - ID1 - Pr - ID2 - Pr- EX1 - Pr - EX2 - Pr - MEM1 - Pr - MEM2 - Pr - WB1 - Pr - WB2 = 0.4+0.1+0.4+0.1+0.4+0.1+0.4+0.1+0.4+0.1+0.4+0.1+0.4+0.1+0.4+0.1+0.4+0.1+0.4 = **4.9ns**

Eventually , it would take **4.9**ns for a single pipeline cycle, in a 10-stage pipeline with each stage split in halves .

G . < Question from textbook >

As per pipeline chart shown in Problem D (for 5-stage pipeline) , after we reach the end of first instruction . Pipeline would be eventually full with instructions operating
At it's highest efficiency of dispatch single instruction at a time or CPI = 1 .
But , from the calculation as we saw in Problem sub-part D , we have 598 cycles cycles spent on execution of
5*(100) = 500 instructions for 100 iterations . Thus at the end it gives us 598 / 500 = **1.196** as CPI . (accounting for hazards) .
Since each pipeline cycle takes around 4.4ns , for 598 cycles it would be around 598*4.4 = **2,631.2 ns** as a total execution time .
Since we have 500 instructions , it would be 2631.2 / 500 = **5.2624 ns** as an average execution time .

Similarly in Problem sub-part E , we are having a 10-stage pipeline split into two stages .
It takes around 609 cycles for the pipeline to complete 500 instructions .
Thus it gives us CPI as 609/500 = 1.218 .
Since each pipeline cycle takes around 4.9ns , for 609 cycles it would be around 609*4.9 = **2,984.1 ns** on an average as an execution time .
Since we have 500 instructions , it would be 2984.1 / 500 = **5.9682 ns** as an average execution time .

C.2]

A . < Question from textbook >

Lets say we have 100 instructions , ideally it would take 100 cycles only without any branch hazard.
Thus we would have 15 conditional branches and 1 unconditional branch .

Out of 15 conditional branches , 9 would be taken and other =6 , would be non-taken branches .
For a worst case scenario , we are assuming that all conditional and unconditional branches are mispredicted .
Thus we would end up losing 3 cycles each for those conditional branches , hence for 15 conditional branches it would be 45 cycles.
And for 1 unconditional branch , 2 cycles would be lost .
Thus with hazards In worst case scenario it would be
100 + 45 (conditional branches resolving cycles) + 2 (unconditional branch resolving cycles) = 147 cycles .
Essentially , Speedup = 100 (cycles without hazards in worst case) / 147 (cycles with hazards) = **0.6802** . Thus machine would be giving a speedup of **0.6802** without any branch hazards .

B . < Question from textbook >

Lets say we have 100 instructions , ideally it would take 100 cycles only without any branch hazard.
Thus we would have 15 conditional branches and 1 unconditional branch .
Out of 15 conditional branches , 9 would be taken and other =6 , would be non-taken branches .
For a worst case scenario , we are assuming that all conditional and unconditional branches are mispredicted .
Thus we would end up losing 10 cycles each for those conditional branches , hence for 15 conditional branches it would be 150 cycles.
And for 1 unconditional branch , 5 cycles would be lost .
Thus the pipeline would take around 100 cycles without any branch hazards , but with hazards In worst case scenario it would be
100 + 150 (conditional branches resolving cycles) + 5 (unconditional branch resolving cycles) = 255 cycles .
Essentially , Maximum speedup = 100 (cycles without hazards in worst case) / 255 (cycles with hazards) = **0.392** without any branch hazards .

C.7]

A . < Question from textbook >

Lets say we have 100 instructions , ideally it would take 100 cycles only without any branch hazard .
Number of branches would be 20 . Among those a mispredicted branch would be only 1 .
For a 5-stage pipeline machine , it gives a stall for every 5 instructions . Thus it would be safe enough to
say that 5-instructions would take around 6 cycles ! . Thus 100 instructions would take around 120 cycles including
Data hazards only !

For a 12-stage pipeline machine , it gives 3 stalls for every 8 instructions . Thus it would be safe enough to
say that 8-instructions would take around 11 cycles ! . Thus 100 instructions would take around (100 + 32 (stalls in execution of those instructions)) = 132
cycles including data hazards only ! .
Total number of cycles needed to execute above 100 instructions = 132 cycles for a 12-stage machine
Without any branch hazards .

Speedup of 12-stage pipeline machine over that (S) = (Number of cycles taken by 5-stage machine) / (Number of cycles taken by 12-stage machine)
Of 5-stage pipeline machine

Thus , S = 120 / 132 = 0.9091 . Looks like 12-stage pipeline machine is slower compared to that of 5-stage pipeline machine .

B . < Question from textbook >

Continuing above example , with 100 instructions . We have one mis-predicted branch for each machine .
Mispredicted branch on a 5-stage machine costs around 2 cycles , Thus 100 instructions would be
100 + 20 (data hazard) + 2 (branch prediction hazard) = 122 . Each clock cycle for a 5-stage machine = 1ns , thus for 122 cycles = **122 ns**
CPI = 122 / 100 = 1.22 ns / instruction

Similarly , mispredicted branch on a 12-stage machine costs around 5 cycles , and Thus 100 instructions would be consuming
100 + 32 (data hazard) + 5 (branch prediction hazard) = 137 / 12 = 12 cycles .
Each clock cycle for a 12-stage machine = 0.6ns , thus for 137 cycles = **82.2 ns**
CPI = 82.2 / 100 = 0.82 ns / instruction

Part E :

The following is x86 assembly. Write equivalent code in C and RISC-V assembly.
(It will probably be easier for you to write the C code first, and then the RISC-V assembly equivalent.)
For C code, include a mapping between the variables you use and the x86 registers; for RISC-V code, include a mapping between the RISC-V registers you use and the x86 registers.
Your code should run on the BRISC-V simulator. Remember to only use the RV32I instruction subset.

Please find equivalent C-code for x86 assembly routine as below : (Mapping of the same
With x86 registers is mentioned In comments) .

```
#define CMP_UPPERLIMIT 99
int main()
{
register int a = 0 , b = 0 , c; // a: eax , b : ebx , c : ecx
while(c<=CMP_UPPERLIMIT) // cmpl $99 , ecx ; jle tloop :
{
a = c; // c : ecx
b = b + a; // ebx = ebx + eax
c = c + 1; // ecx = ecx + 1
}
return 0;
}
```

RISC-V assembly routine as below :

```
main:
addi sp,sp,-32
sd s0,24(sp)
addi s0,sp,32
sw zero,-28(s0)
sw zero,-20(s0)
.L3:
lw a5,-24(s0)
sext.w a4,a5
li a5,99
bgt a4,a5,.L2
lw a5,-24(s0)
sw a5,-28(s0)
lw a4,-20(s0)
lw a5,-28(s0)
addw a5,a4,a5
sw a5,-20(s0)
lw a5,-24(s0)
addiw a5,a5,1
```

```
sw a5,-24(s0)
j .L3
.L2:
li a5,0
mv a0,a5
ld s0,24(sp)
addi sp,sp,32
jr ra
```

Register mapping for RISC-V and x86 is as below :

A5 register : %ecx , %ebx and %eax.
A4 : %eax ,

Part F :

The CDC6600 implemented a Scoreboard to enable dynamic scheduling in the datapath.
An alternative mechanism, called the Tomasulo Algorithm was implemented by IBM on the System/360 Model 91.
Provide detailed descriptions of these two mechanisms and discuss how they resolved different kinds of hazards.
Argue for which mechanism you would choose to implement in practice and defend your choice.
Identify a design that uses at least one of these mechanisms today

Scoreboard Scheduler

Scoreboard mechanism implemented on CDC6600 is a dynamic scheduler in datapath . Basically , it converts a 3-staged pipeline **ID , EX , MEM** to **Issue , Read Operands , Execute , Write Result .**
Issue stage : It would wait for a functional unit as well as register associated with instructions to be free
Scoreboard would eventually update it's internal data structure . This updation helps when there is a **WAW** hazard , instruction issue is stalled till the hazard is cleared .
Once an instruction is giving a green signal to being issued , it goes forward to **read operands stage** .
In this stage , source operands used in instruction would be checked if there are any writes active to those . It would put a stall in case of a **RAW hazard** , thereby helping to resolve it .
Execution doesn't provide any stall and post completion it notifies scoreboard about completion .
Write result stage , would provide a stall if there is an ongoing read operation active on the register which would be overwritten thereby helping resolve **WAR** hazard .

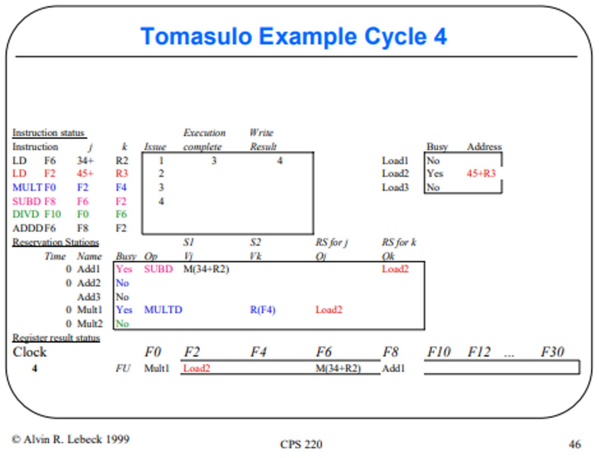
Example below :

```
DIVD F0,F2,F4
ADDD F10,F0,F8
SUBD F8,F8,F14
```

SUBD will wait for ADDD to complete .
However a big drawback of above mechanism is that it resolves hazards by putting stalls rather than using forwarding of states .

Tomasulo Algorithm :

Above Drawbacks of absence of register forwarding and tries to resolve hazards by using reservation stations and an instruction queue .
When an instruction is fetched from an instructional unit for functioning purpose , it would check it there is any reservation station free or not (Every functional unit would have a reservation station which acts to cater various different types of instructions with control buffers like Vj , Vk (values of source operands) , Qj , Qk (registers of source operands) , Rj , Rk (register ready flag for source operands) , Busy : indicating reservation station and Functional unit is busy .) For any instruction say load , a load operator would be assigned and a destination control buffer which would be further read by subsequent operations rather than it waiting for value to be ready uptight in the register itself . It basically performs a register renaming , with intermediate values in control loop buffers that helps to resolved WAR , RAW and WAW hazards rather than providing a stall .



Tomasulo Example Cycle 4

Instruction status

Instruction		j	k	Issue	Execution complete	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2		
MULT	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	Yes	45+R3
Load3	No	

Reservation Stations

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for j Qj	RS for k Qk
0	Add1	Yes	SUBD	M(34+R2)			Load2
0	Add2	No					
	Add3	No					
0	Mult1	Yes	MULTD		R(F4)	Load2	
0	Mult2	No					

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
4	FU	Mult1	Load2		M(34+R2)	Add1				

© Alvin R. Lebeck 1999 CPS 220 46

Please find above example which demonstrates a reservation station of a Tomasulo mechanism .
Thus Tomasulo Algorithm stands as a winner against scoreboard scheduler owing to it's feature of register renaming dynamically
To resolve hazards rather than having a stall in between them .
Decentralization of Tomasulo Algorithm against that of centralization of scoreboard helps to further increase the speed of deliverables

PowerPC is a best example of current day processor which uses Tomasulo Algorithm for scheduling purpose .

Extra Credit :

Find a tool to generate a data address trace for the fibo.c program run on X86. Only collect data addresses execution.
Identify how many unique pages of memory (assume 4KB pages) are touched by data references in your code.
How are you going to get this done? You could modify a simulator that can save the trace, or you can find a customized tool
 that provides you with the ability to generate the trace and records addresses for loads and stores. This is your choice.
Provide details on what tool you used

We are using an intel PIN tool , for profiling fib.c file .
Please find the steps of setting up pin :
1 . Generate an image of fib.c file (using compilation step : g++ fib.c -o fib.so)
2 . Use above output file or executable as an image that needs to be JITted by PIN .

3 . Write a pin program that would help to only trace data address traces , followed by an algorithm
which would only push distinct 4KB page accesses to an array and also maintain a track of the pages being pushed .
Maintain a state counter that would track number of **memory reads , writes , execution of fibonacci routine
Has started , routine is completed and distinct 4KB page accesses** .

Code snip below , Registers **Instruction** function as a callback for any new instruction and **ImageLoad** function when an
Image is being loaded .

```
<snip>
int main(int argc, char * argv[])
{

  // Initialize PIN module for instrumentation purpose
  if (PIN_Init(argc, argv)) return Usage();

  // OPen a file in output mode for writing purpose
  OutFile.open(KnobOutputFile.Value().c_str());

  PIN_InitSymbols();

  // Instruction function would be executed when a new instruction is
  // executed
  INS_AddInstrumentFunction(Instruction, 0);

  // Register ImageLoad to be called when an image is loaded
  IMG_AddInstrumentFunction(ImageLoad, 0);
</snip>
```

4 . ImageLoad function creates an event that triggers a callback on the start and exit of Routine (RTN) execution : Fibonacci .
 Callbacks initiate the state machine counter , to start writing data address traces to output file in **Instruction** function . Please
find snips below for the same .

```
<snip>
#define FIBONACCI_MAIN "_Z9Fibonaccim"

VOID ImageLoad(IMG img, VOID *v)
{

    RTN fibonacciwrapperRtn = RTN_FindByName(img, FIBONACCI_MAIN);
    if (RTN_Valid(fibonacciwrapperRtn))
    {
        OutFile << " Fibonacci Hit ! " << endl;
        RTN_Open(fibonacciwrapperRtn);

        // Instrument fibnacci_wrapper() to print the start in trace.
        RTN_InsertCall(fibonacciwrapperRtn, IPOINT_BEFORE, (AFUNPTR)fibonacci_wrapper_start,
                                              IARG_ADDRINT, FIBONACCI_MAIN,
                                              IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                                                    IARG_END);
        RTN_InsertCall(fibonacciwrapperRtn, IPOINT_AFTER, (AFUNPTR)fibonacci_wrapper_exit, IARG_FUNCRET_EXITPOINT_VALUE, IARG_END);
        RTN_Close(fibonacciwrapperRtn);
    }
}

VOID fibonacci_wrapper_start()
{
  fibonacciRecursiveFunctionTracker = fibonacciRecursiveFunctionTracker + 1;
  OutFile << " Fibonacci Start ! " << endl;
}

VOID fibonacci_wrapper_exit()
{
  fibonacciRecursiveFunctionTracker = fibonacciRecursiveFunctionTracker - 1;
  OutFile << " Fibonacci Exit ! " << endl;
}

std::vector<UINT64> DistinctInstruction4KBpages ;

// Push a Distinct 4KB Page onto vector or container
void DistinctInstruction4kbPagePush(UINT64 addr)
{
  DistinctInstruction4KBpages.push_back(addr);
  int last_index = DistinctInstruction4KBpages.size()-1;
  int iter_index = last_index ;

  while((iter_index !=0) && (DistinctInstruction4KBpages[iter_index] <  DistinctInstruction4KBpages[iter_index-1]))
  {
    UINT64 temp =  DistinctInstruction4KBpages[iter_index];
    DistinctInstruction4KBpages[iter_index] = DistinctInstruction4KBpages[iter_index-1];
    DistinctInstruction4KBpages[iter_index-1] = temp;
    iter_index = iter_index - 1;
  }

  if( iter_index > 0 )
  {
    if(DistinctInstruction4KBpages[iter_index] ==  DistinctInstruction4KBpages[iter_index-1])
    {
      DistinctInstruction4KBpages.erase(DistinctInstruction4KBpages.begin()+iter_index);
    }
    else
    {
      OutFile << " Instruction Distinct Page Added ! " << endl;
    }
  }
}
```

</snip>

Results are quite expected as fib.c is using only a single page for entire program run
For 1000 iterations , please find the results as below

```
Memory write count = 51514136
Memory read count = 56197307

Min instruction address = 140735450942696 -> 7FFF 868F 8CE8
Max instruction address = 140735450944152 -> 7FFF 868F 9298

4kb Distinct Memory Page access count = 1

Page : 7FFF 868F 74EC ( Start address )
```

As the fibonacci program (recursive version) doesn't have a high working set it Is plausible that it would use only
A single page for all of it's run and try to use more and more stack for it's recursive operation . It maybe possible
That it's iterative version may end up using more pages and can cause page thrashing too , if the working set
Grows .

Few Keypoints :
1 . We are considering only memory accesses , by using INTEL PIN tool to filter the same thereby excluding stack accesses
And instruction trace . Thus results that are enclosed in the report are pursued keeping in mind only memory accesses only .
2 . Above <snips> are C-program which are written by modifying the itrace program of Intel PIN .
    Modifications include :
    1 . Invoke Tracing on Start of Fibonacci api (so that we don't consider shared libraries accesses)
    2 . Filter-in only memory accesses .
    3 . Check on which page does the memory access belong to .
    4 . Push a distinct memory page access to vector containers as shown in C-programs above .