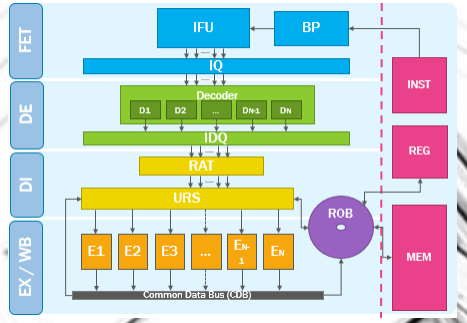
University of BRISTOL

Superscalar Processor Simulator

Felipe Galindo Sanchez

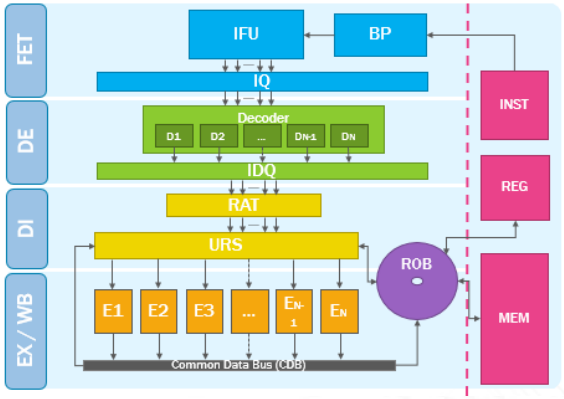
December 2015

COMSM0109 – Advanced Computer



Superscalar Processor Simulator

- Simulator
  - Getting Started
  - Debugging
  - Compiler (Assembly to Machine code)
- Processor Architecture
  - Diagram
  - Features
  - Configurable
  - Instructions Set
- Benchmarks
  - Programs
  - Experiments



Simulator - Getting Started

GitHub Repository
github.com/felipegs01/processor_simulator

Just type **simulator.exe –help** or **./simulator --help** to list the help with all the commands and flags available, otherwise **run** is the main command.

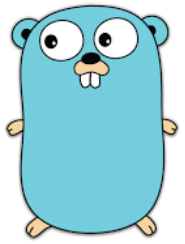
```
USAGE:
  simulator run [command options] <assembly-filename>
OPTIONS:
  -s, --step-by-step        Run interactively step by step
  -v, --verbose             Verbose on debug mode
  -o, --output-folder       Output folder where to store debug and memory files
  -c, --config-filename     Processor config filename path with the processor architecture config
  --max-cycles              Maximum number of cycles to execute
```

Sample:

```
run samples/programs/fibonacci.asm -c samples/configs/default.config -o results/my-test --max-cycles 1000 --step-by-step -v
```

Developed in Go

https://golang.org/



OS Compatible



Simulator – Debugging

During simulation - Interactive

If the program is executed using the flag ***-s*** or ***--step-by-step***, the state of registers and/or data memory can be seen on each cycle that is being executed.

```
Press the desired key and then hit [ENTER]...
 - (R) to see registers memory
 - (D) to see data memory
 - (E) to exit and quit
 - (*) Any other key to continue
```

If selected `R` or `D`, the data will be displayed in the following format:

	0x00	0x04	0x08	0x0C
0x00	0x0000000A	0x0010000A	0x000C0000	0x00000000
0x10	0x000100E8	0x00000008	0x00000012	0x00000087
0x20	0x0000FF00	0x00000000	0x00D00068	0x002000A8
0x30	0x000000E8	0x0000C008	0x00000012	0x00000087
0x40	0x00000012	0x00000000	0x00100000	0x00000000

At the end of simulation - Persisted files

At the end **six files** will be generated with details of the execution, debugging and final memory states.

The location of those output files can be selected with the flag ***-o*** or ***--output-folder***

- **assembly.hex**: Machine code interpreted by the processor
- **memory.dat**: Final state of the data memory.
- **registers.dat**: Final state of the registers.
- **output.log**: Execution resources according to the configuration and output statistics.
- **debug.log**: Complete log for debugging purposes.
- **pipeline.dat**: Pipeline diagram of the different executed instruction stages vs execution cycles

Simulator – Compiler (Assembly to Machine code)

This application has a builtin translator that converts *human readable assembly* instructions into *machine code*, the available instructions allowed are the ones defined on the previous instructions section.

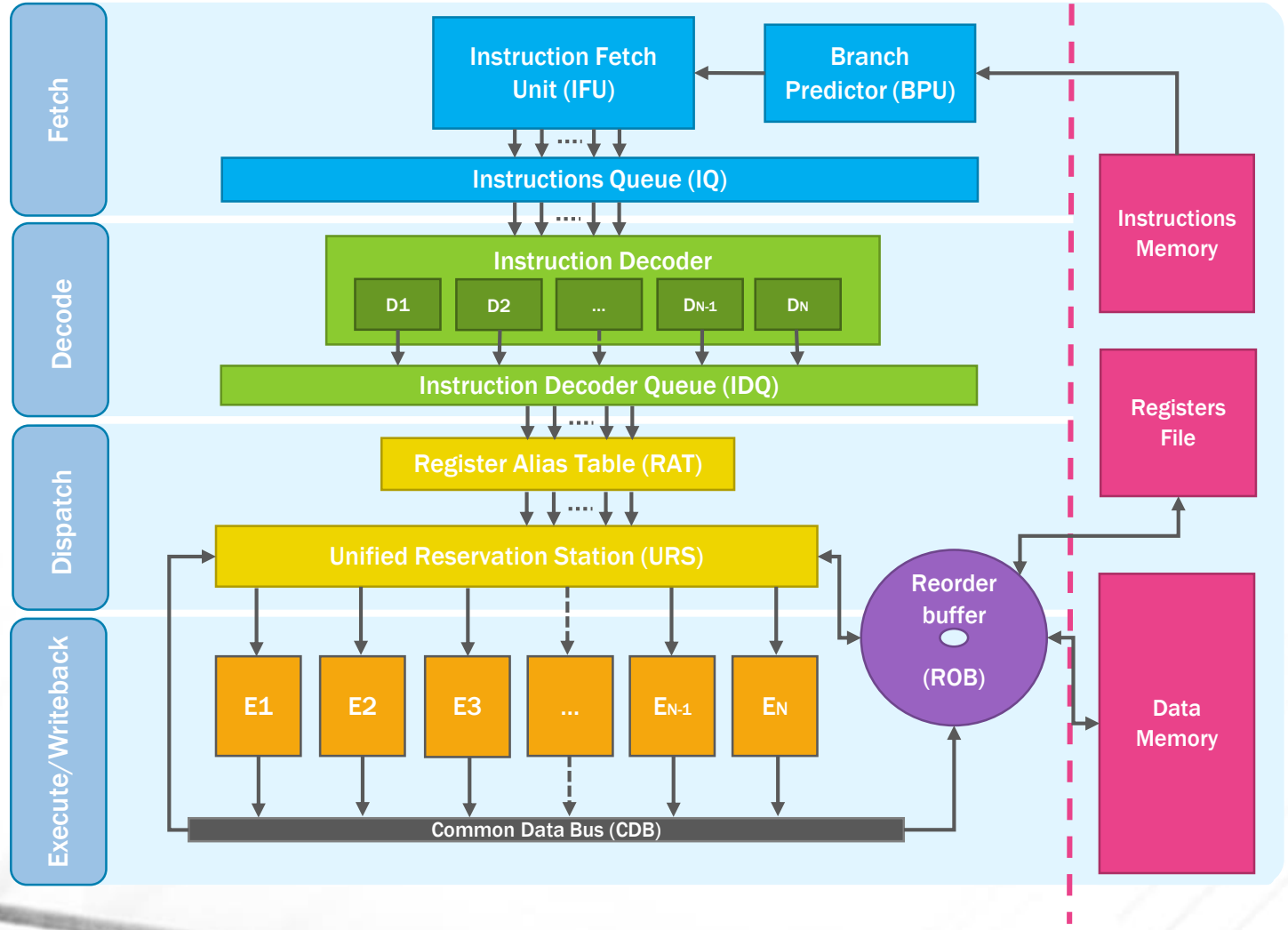
- Only one instruction allowed per line
- Comments prefix is `;`
- Comments are allowed to be on a single line or after an instruction in the same line
- It does not care about the amount of empty spaces or tabs
- Branch labels must be on a single line
- No instructions allowed to be on the same line where the branch label is declared
- Blank lines are allowed

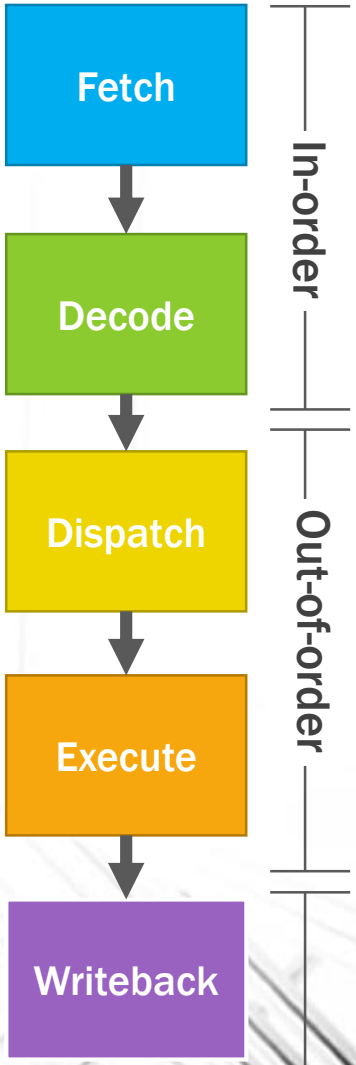
Pre-fill data memory

```
@0x01E0: 09 0D 23 0A 07 18 0C 06 15 0F   ; Macro to pre-fill some data in memory before the program
@0x02E0: 49 2E 05 EA C6 06 3C 15 0F 41   ; is executed at an specific memory address

PROCESS_LOOP:                            ; Here is a label followed by an inline comment
ADDI    R1, R1, 1                        ; Here is a instruction along with its operands and an inline comment
                                         ; Here it is an empty line which is allowed followed by an inline comment
ADD     R15, R15, R16                    ; R15 += C[I]
BLT     R1, R20, PROCESS_LOOP            ; Here is an instruction using a branch label followed by an inline comment
```

Processor Architecture - Diagram





Processor Architecture – Features (Sample)

Overview

- 32 bits architecture
- Scalar, Pipelined or N-way superscalar
- Out-of-order execution and non-blocking issue
- 32 general purpose registers (32-bit) (used for integer & FP)
- 1 MB Instructions Memory
- 1 MB Data Memory

Five-stage pipeline

- Fetch, Decode, Issue/Dispatch, Execute & Writeback

Execution units (EU's)

- 2 ALU units
- 2 Load/Store units
- 1 Branch units
- 1 FPU units

Branch Prediction

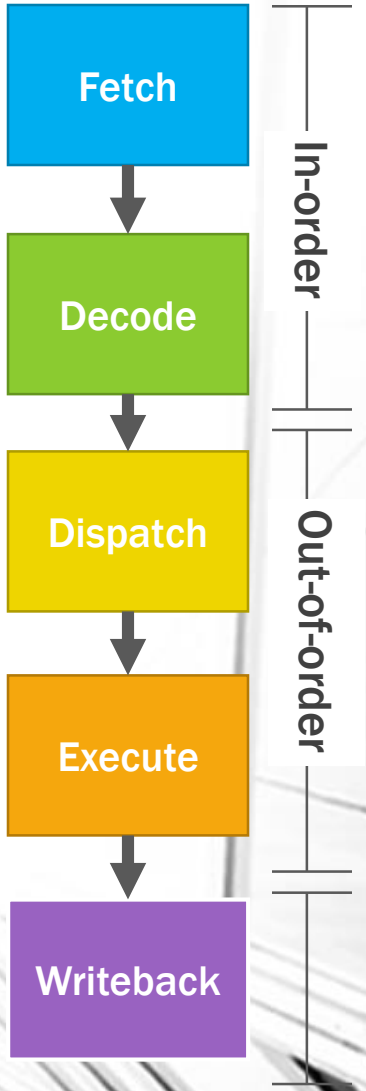
- None (Stall)
- Static: Always, Never, Forward, Backward
- Dynamic: One bit predictor, Two-bit predictor (BHT)

Front-End Pipeline (In-order)

- Instruction Fetch Unit (IFU):
  - *16 bytes fetch on each cycle (4 instructions)*
- Instruction Queue (IQ):
  - *18 instructions buffer*
- Instruction Decoding Unit (IDU)
  - *4 decoding units*
- Instructions Decoded Queue (IDQ):
  - *28 instructions buffer*

Execution Pipeline (Out-of-order)

- Common Data Bus (CDB)
- Register Renaming
  - *Register Alias Table (RAT) with 32 entries*
- Reorder buffer (ROB)
  - *32 entries*
  - *Up to 4 instructions written back on each cycle*
- Unified reservation station (URS):
  - *128 entries*
  - *Up to 6 instructions dispatched on each cycle*



Processor Architecture - Configurable

Overview

- 32 bits architecture
- Scalar, Pipelined or N-way superscalar (choose)
- Out-of-order execution and non-blocking issue
- N general purpose registers (32-bit) (used for integer & FP)
- N MB Instructions Memory
- N MB Data Memory

Five-stage pipeline

- Fetch, Decode, Issue/Dispatch, Execute & Writeback

Execution units (EU's)

- N ALU units
- N Load/Store units
- N Branch units
- N FPU units

Branch Prediction (choose predictor)

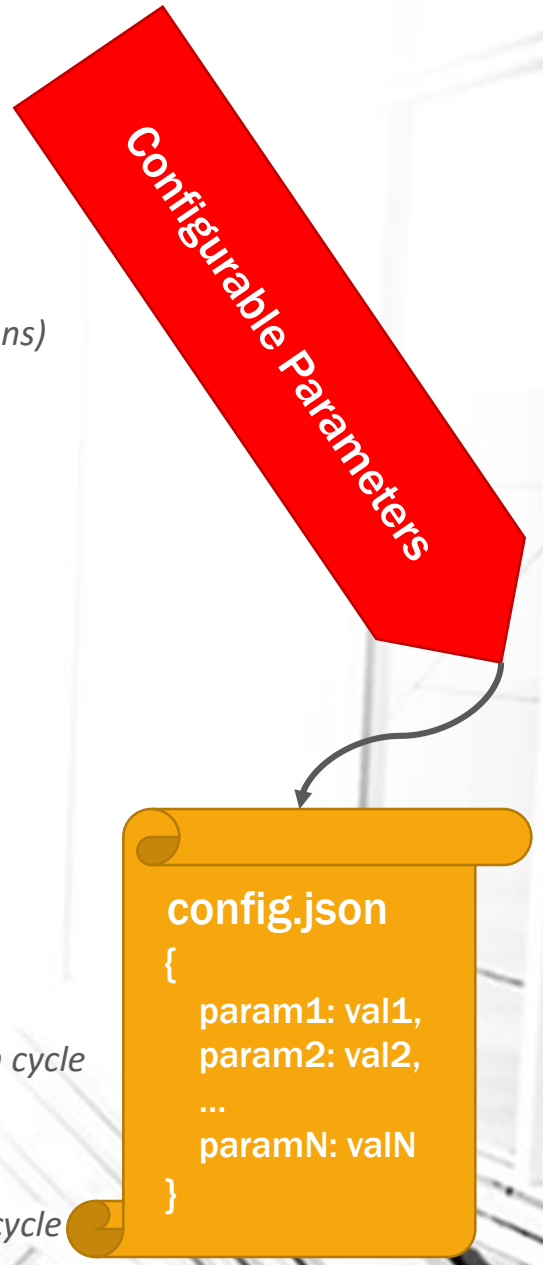
- None (Stall)
- Static: Always, Never, Forward, Backward
- Dynamic: One bit predictor, Two-bit predictor (BHT)

Front-End Pipeline (In-order)

- Instruction Fetch Unit (IFU):
  - *N bytes fetch on each cycle (N instructions)*
- Instruction Queue (IQ):
  - *N instructions buffer*
- Instruction Decoding Unit (IDU)
  - *N decoding units*
- Instructions Decoded Queue (IDQ):
  - *N instructions buffer*

Execution Pipeline (Out-of-order)

- Common Data Bus (CDB)
- Register Renaming (enabled/disabled)
  - *Register Alias Table (RAT) with 32 entries*
- Reorder buffer (ROB)
  - *N entries*
  - *Up to N instructions written back on each cycle*
- Unified reservation station (URS):
  - *N entries*
  - *Up to N instructions dispatched on each cycle*



Processor Architecture - Instructions Set

- 32 bit instructions wide
- Instructions formats: R, I & J
- Instructions types: *Arithmetic (ALU & FPU), Load/Store, Control/Branch*

Arithmetic/Logic (ALU & FPU)

EU	Syntax	Description	Type
ALU	add/addi Rd,Rs,Rt/C	Rd = Rs + Rt/C	R
	sub/subi Rd,Rs,Rt/C	Rd = Rs – Rt/C	R
	cmp Rd,Rs,Rt	Rd = Rs <=> Rt	R
	mul Rd,Rs,Rt	Rd = Rs * Rt	R
	shl/shli Rd,Rs,Rt/C	Rd = Rs << Rt/C	R
	shr/shri Rd,Rs,Rt/C	Rd = Rs << Rl/C	R
	and/andi Rd,Rs,Rt	Rd = Rs & Rt/C	R
	or/ori Rd,Rs,Rt/C	Rd = Rs | Rl/C	R
FPU	fadd Rd,Rs,Rt	Rd = Rs + Rl	R
	fsub Rd,Rs,Rt	Rd = Rs – Rl	R
	fmul Rd,Rs,Rt	Rd = Rs * Rl	R
	fdiv Rd,Rs,Rt	Rd = Rs / Rl	R

Formats

Type R	**Format (32 bits)**				
	Opcode (6)	Rd (5)	Rs (5)	Rt (5)	Not used (11)
Type I	**Format (32 bits)**				
	Opcode (6)	Rd (5)	Rs (5)	Immediate (16)	
Type J	**Format (32 bits)**				
	Opcode (6)	Address (26)			

Load/Store

Syntax	Description	Type
lw Rd,Rs,C	Rd = M[Rs + C]	I
sw Rd,Rs,C	M[Rs + C] = Rd	I
lli Rd,C	Rd = C	I
sli Rd,C	M[Rd] = C	I
lui Rd,C	Rd = C << 16	I
sui Rd,C	M[Rd] = C << 16	I

Control/Branch

Syntax	Description	Type
beq Rd,Rs,C	br on equal	I
bne Rd,Rs,C	br on not equal	I
blt Rd,Rs,C	br on less	I
bgt Rd,Rs,C	br on greater	I
j C	jump to C	I

Benchmarks

REG	Registers interaction
FPO	Floating Point Operations
MEM	Memory interaction
HAZ	Data Hazards
BRN	Conditional/Unconditional branches
LOP	Loop

Program	Flavors	REG	FPO	MEM	HAZ	BRN	LOP	Special targeted features
Load/Store & ALU	Single	✓	X	X	X	X	X	EU's throughput, Ideal N-way execution
Arithmetic Oper.	Single	✓	✓	X	X	X	X	ALU & FPU correctness
Fibonacci	Single	✓	X	X	✓	✓	✓	Register Renaming
Factorial	Single	✓	X	X	✓	✓	X	Register Renaming
Hazards	RAW, WAW, WAR	✓	X	X	✓	X	X	Reorder buffer
Loop Vectors	Forward jumps, Backward jumps	✓	X	✓	✓	✓	✓	BR Predictors
Inner Product	Single	✓	X	✓	✓	✓	✓	Register Renaming, RS
Bubble Sort	Forward jumps, Backward jumps	✓	X	✓	✓	✓	✓	Branch Predictor, Register Renaming, RS

Experiments

Experiments and measurements are averaged over ~ 5 executions in order to discriminate branch guesses and non-deterministic events

Experiment 1 – Scalar, Non-Pipelined & Superscalar

Features targeted

Pipeline | Scalar | Superscalar | Interferences

Hypothesis

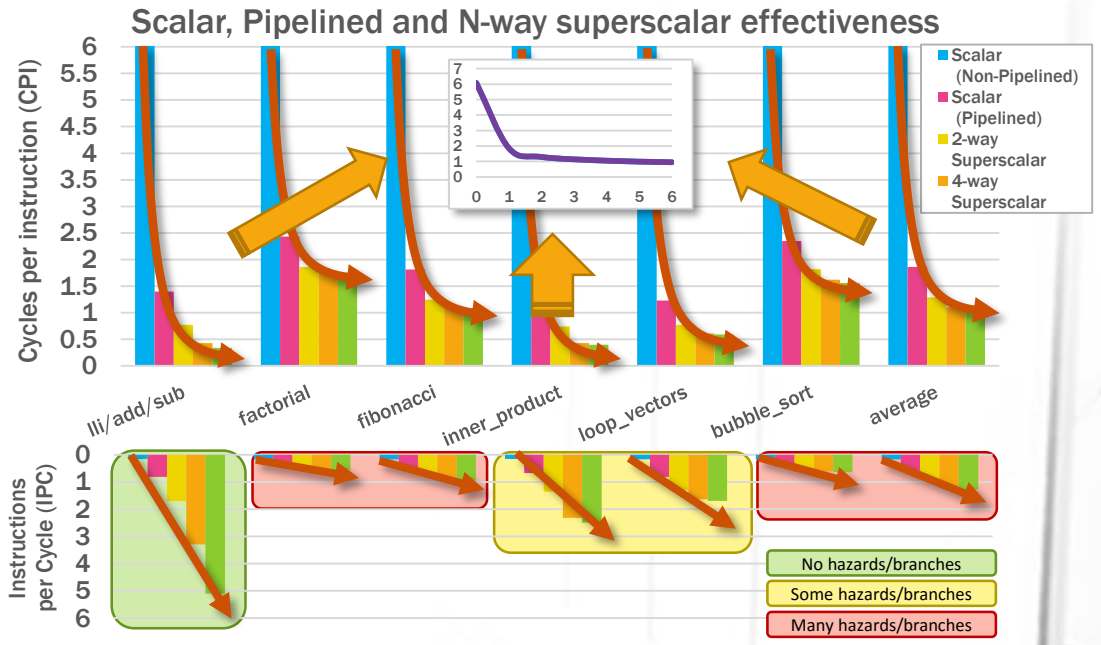
- In a **scalar non-pipelined** processor the **CPI** will be close to the **pipeline-size** (not exact due to branches)
- In a **pipelined** processor (from **1-way scalar** to **N-way superscalar**) the improvements in **CPI** will **increase**.
- In **average benchmarks/programs** including branches, hazards and non-deterministic events and the program itself it will take **one cycle per instruction.**

Experiment

Drive all benchmarks and measure the number of cycles per instruction (CPI) for the following processor architectures:

- Scalar (non-pipelined)
- Scalar pipelined
- 2-way, 4-way, and 6-way superscalar

Result



- **Pipelined** and **superscalar** processors definitely **improve** the **CPI/IPC**.
- **IPC** is **linearly proportional** to the **N-way** of a **superscalar.**
- The **improvement ratio** (CPI/IPC) will **depend** on **hazards** and **branches** in an executed program.
- Programs with **no hazards/branches** (e.g *lli/add/sub*) will behave close to processing **N instructions per cycle** for a **N-way superscalar** given.

Experiment 2 – Number of Execution Units (EU's)

Features targeted

Execution Units

Hypothesis

- Incrementing the # of execution units will execute programs faster.
- After a number of execution units, the execution time reduced won't be that significant.

Experiment

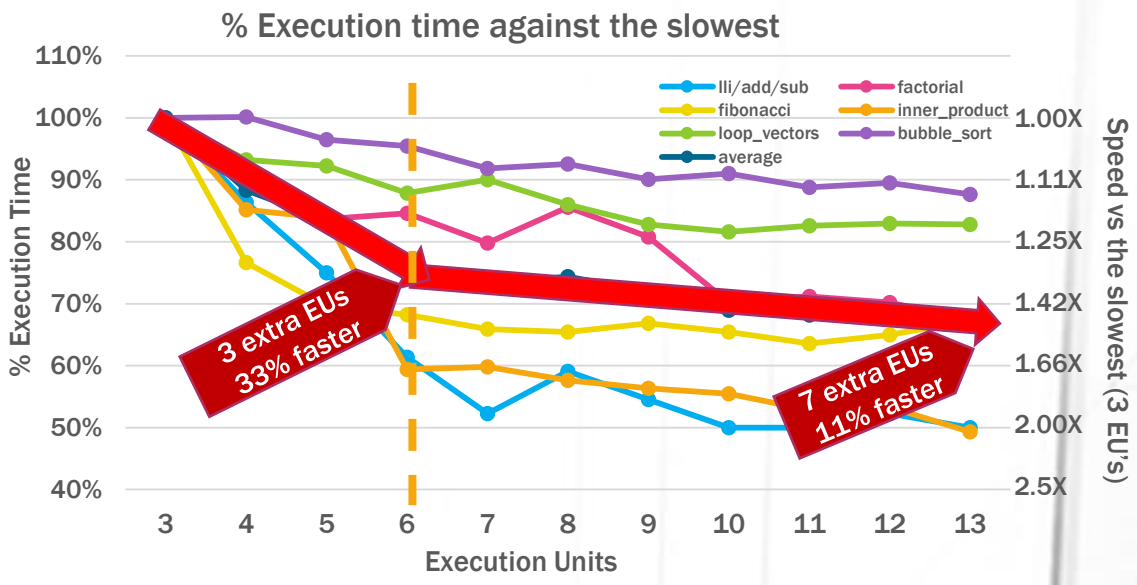
Drive the different available benchmarks and measure the number of cycles executed with configurations containing 3 EU's (1 branch, 1 Load/Store & 1 ALU) up to 13 EU's

Obtain the percentage of time consumed against the slowest configuration (3 EU's)

$$\%\ exec\ time\ (\#EU's) = \frac{\#\ cycles\ executed\ (\#\ EU's)}{\#\ cycles\ executed\ (3\ EU's)}$$

Find the # EU's where the % savings is not significant anymore.

Result



- From **3 up to 6 EU's**, programs are **11% faster per** each **EU**
- From **7 up to 13 EU's**, programs are **1.57% faster per** each **EU**
- Therefore it's concluded that incrementing the # of EU's will improve the execution time with a **significant % of savings up to ~6 EU's**, but more than that, the % of savings won't be that significant

Experiment 3 – Branch Predictors and miss-prediction

Features targeted

Fetch Unit

Branch Predictors

Hypothesis

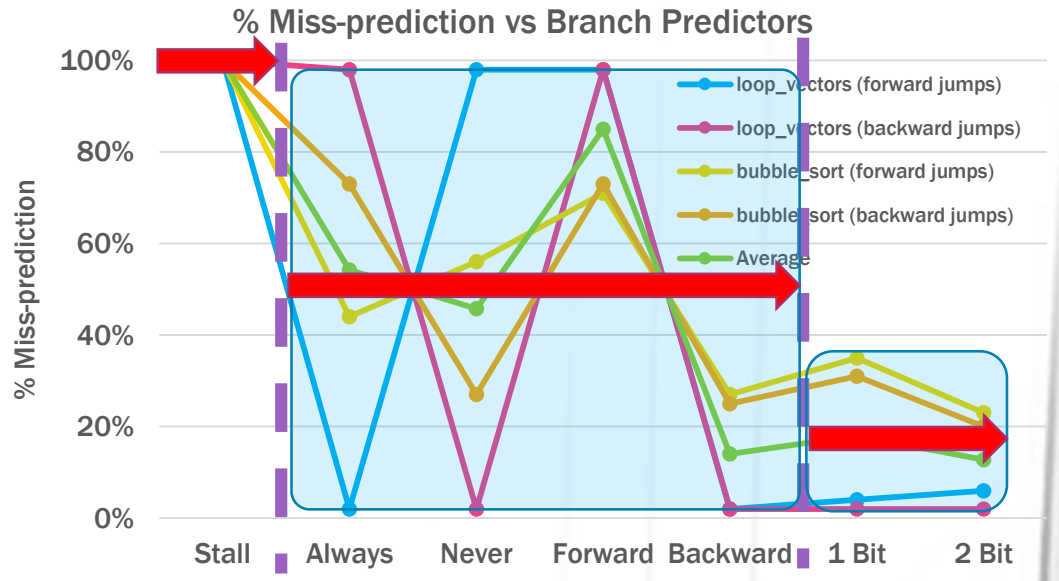
- Branch predictors improves the execution time of programs
- Static predictors can perform good or bad depending on how programs are coded with a high variability
- Dynamic predictors perform better no matter which way programs are coded with a lower variability.

Experiment

Drive benchmark programs coded with two flavors: *backward jumps* and *forward jumps*, in order to benchmark the different results with different programing ways

Compare benchmarks and flavors with the different branch predictors from *none*, *static* to *dynamic* and measure the percentage of miss prediction -> (the more miss prediction, the more time it will take to execute a program)

Result



- We can see that **static predictors** have a bigger miss prediction variability range depending on how the program is coded (conditional branches)
  Approx. **50% miss prediction** with **+/- 50%** variability
- **Dynamic predictor** performed better no matter what flavor was executed
  Approx. **20% miss prediction** with **+/- 10%** variability

Experiment 4 – Reorder Buffer Size

Features targeted

Reorder Buffer | Reservation Station

Hypothesis

- Due to ROB's capacity, some instructions will be stalled on dispatching if the ROB is full (meaning there are pending results to be committed), therefore incrementing the # of entries in the buffer, the speed will be incremented.
- The speed increment will stop at a certain # of ROB entries.

Experiment

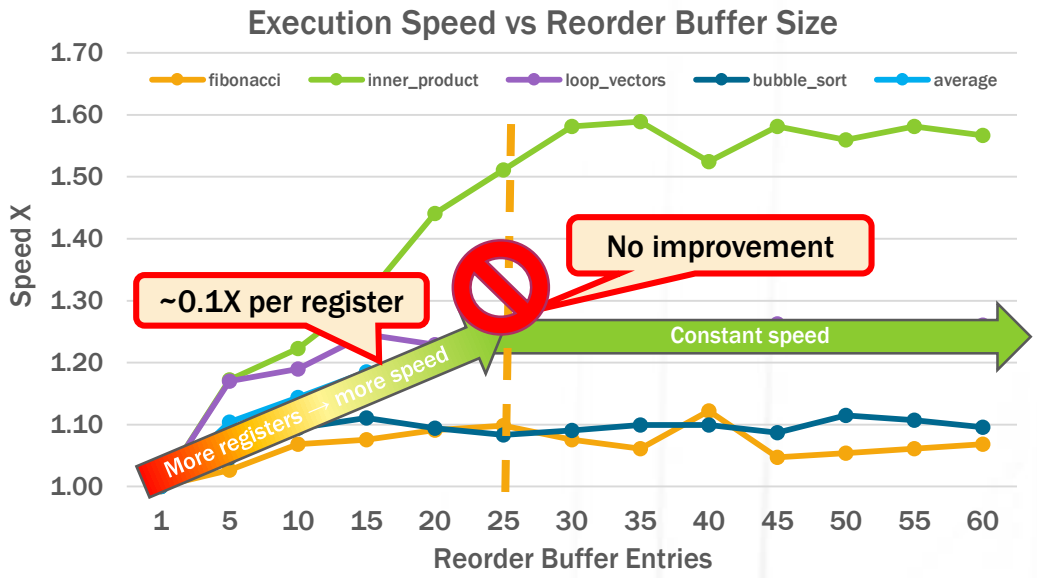
Drive the different available benchmarks and measure the number of cycles executed with configurations containing a ROB starting from 1 entry up to 60 entries.

Obtain the speed as a function of the # of entries against the slowest (1-entry ROB)

$$speed\ (ROB\ size) = \frac{\#\ cycles\ executed\ (\text{1-entry ROB})}{\#\ cycles\ executed\ (ROB\ size)}\ X$$

Find the # of ROB entries where the increment stops.

Result



- From a **1-entry** ROB to a **~25-entry** ROB the speed will be incremented approximately **1.24X** or **24%.** (**1.1% or 1.1X** for **each** added **register**)
- ROBs with more than **~25 entries**, (unless programs with mixed hazards provoking to have more than ~25 dispatched instructions and pending to be committed) **won't make** any **improvement** at all.

Experiment 5 – Register re-naming using RAT

Features targeted

Register Re-naming | Register Alias Table (RAT)

Hypothesis

- Using *register re-naming* will avoid stall WAR and WAV register dependencies
- Once *register re-naming* enabled, the RAT size will improve the execution speed until an approximated RAT size when the speed improvement won't be that significant.

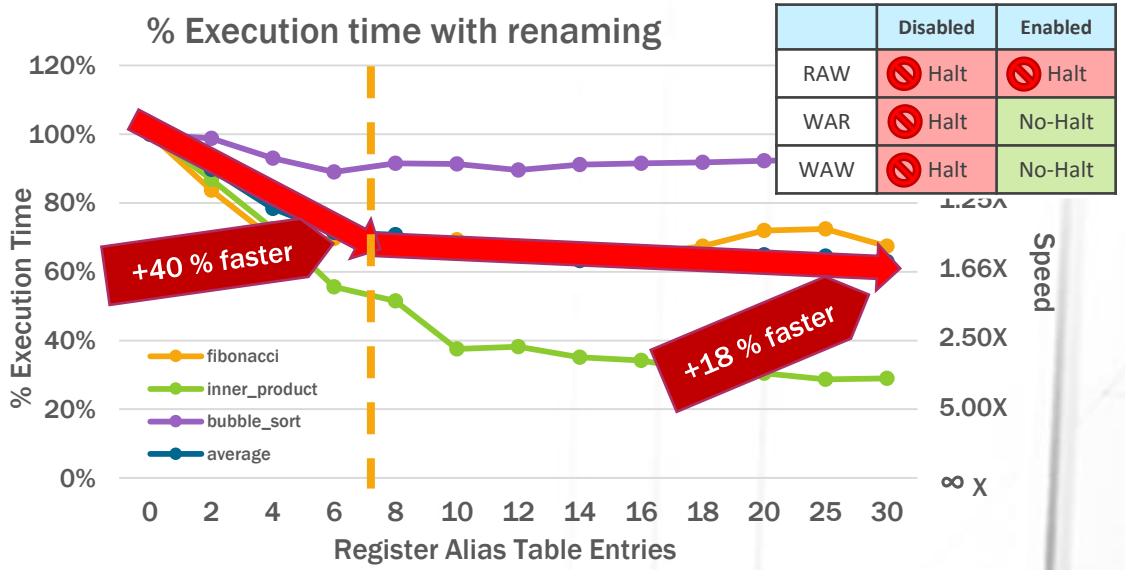
Experiment

1) Drive RAW, WAR and WAW hazards with and without *register re-naming* enabled. Then with the pipeline diagram determine if the dependency was or was not halted.

2) Drive available benchmarks and measure percentage of execution time against *register re-naming* disabled

$$\%\ exec\ time\ (RAT_{size}) = \frac{\#\ cycles\ executed\ (RAT_{size})}{\#\ cycles\ executed\ (RR_{disabled})}$$

Result



- Using ***register re-naming*** definitely improves the performance, **avoiding halting** on **WAR** and **WAW** dependencies by using other physically registers in the **Register Alias Table (RAT)**.
- Using a **3 to 7-entries RAT**, programs are faster approximately in **40%.** (**5.7%** faster for **each** added **entry** to the RAT)
- Using a **7 to 30-entries**, programs can be **18%** faster (not that significant). **0.8%** faster for **each** added **entry** to the RAT)