

Advanced Computer Architecture

—

Part I: General Purpose Trimaran Practical Lab (I)

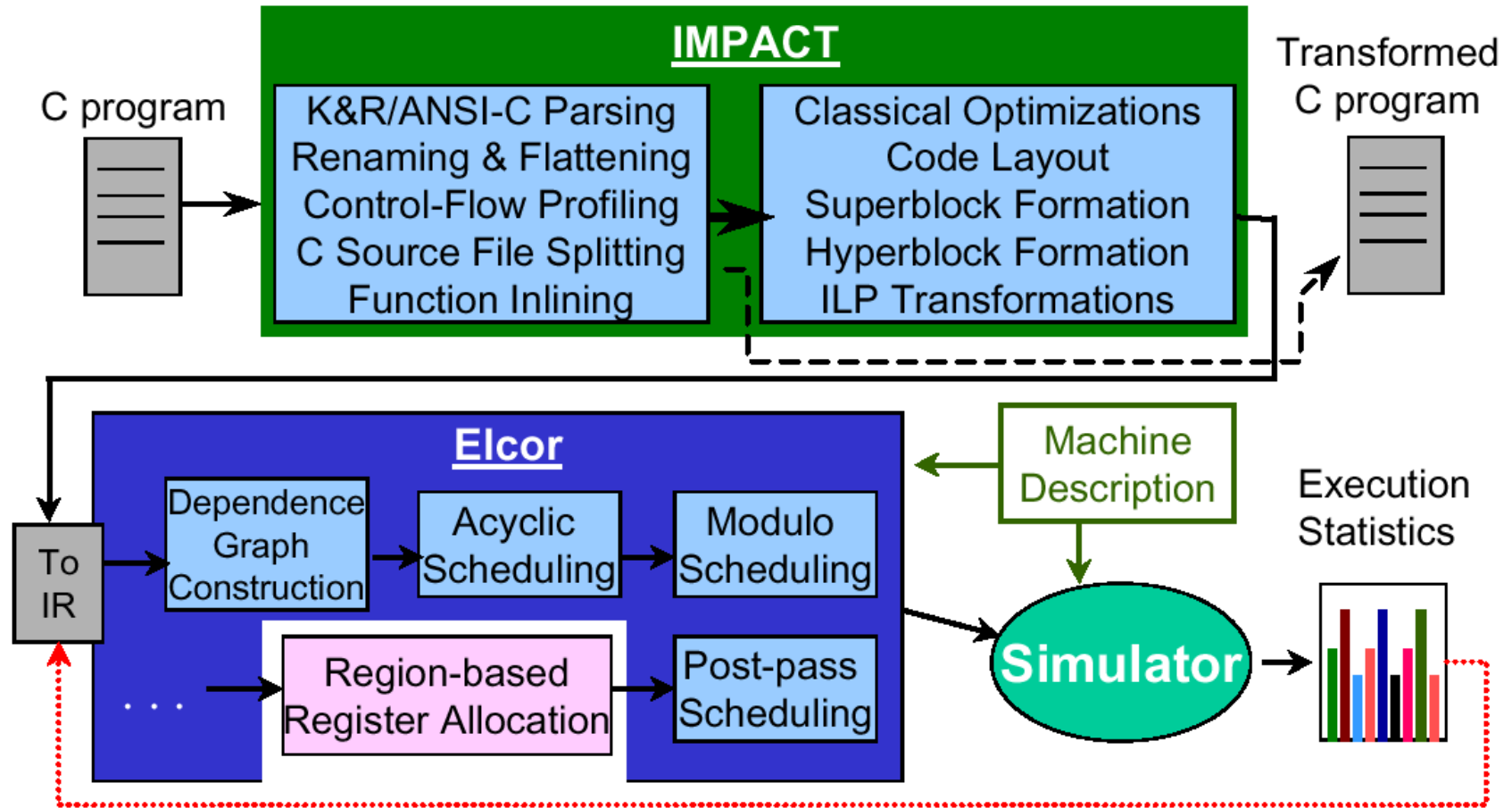
EPFL – I&C – LAP

Trimaran

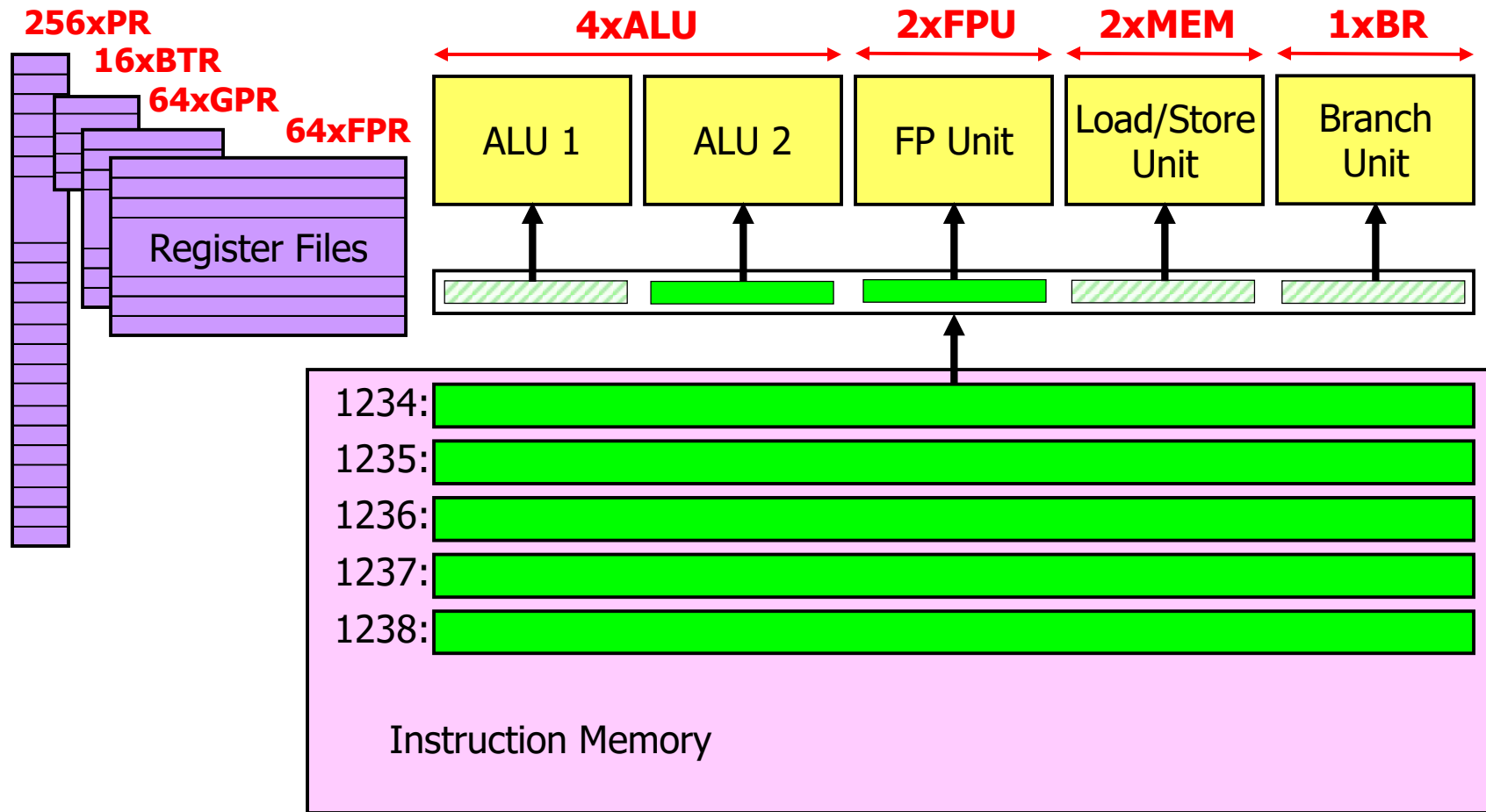
- ❑ VLIW ¹compiler and ²simulator
- ❑ Based on HPL-PD parameterized processor architecture
- ❑ Allows various parameter code optimizations
- ❑ Provides a visualization of code generation

- ❑ Our goal is to study the effects of compiler optimizations on VLIW performance

Trimaran Organization



Processor Architecture



Processor Configuration

- ❑ HPL-PD processor is configured by a config file

`~/trimaran-workspace/machines/hpl_pd_elcor_std.hmdes2`

- ❑ Configure

- ❖ Number of functional units and their latencies
- ❖ Number of registers available for
 - Computation (general purpose and floating point)
 - Compiler optimization (predication and branch targets)

- ❑ In this lab, we will only focus on compiler optimization

Trimaran Interface

- ❑ Start T-shell/C-shell

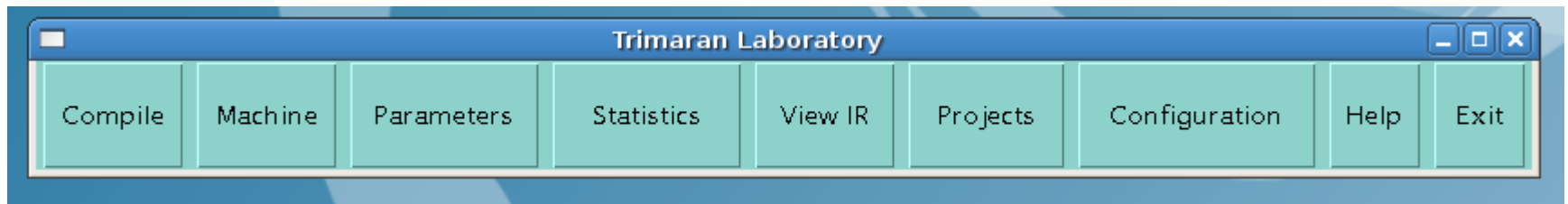
```
tcsh
```

- ❑ Launch Trimaran

```
tri &
```

- ❑ Working directory is generated on first launch

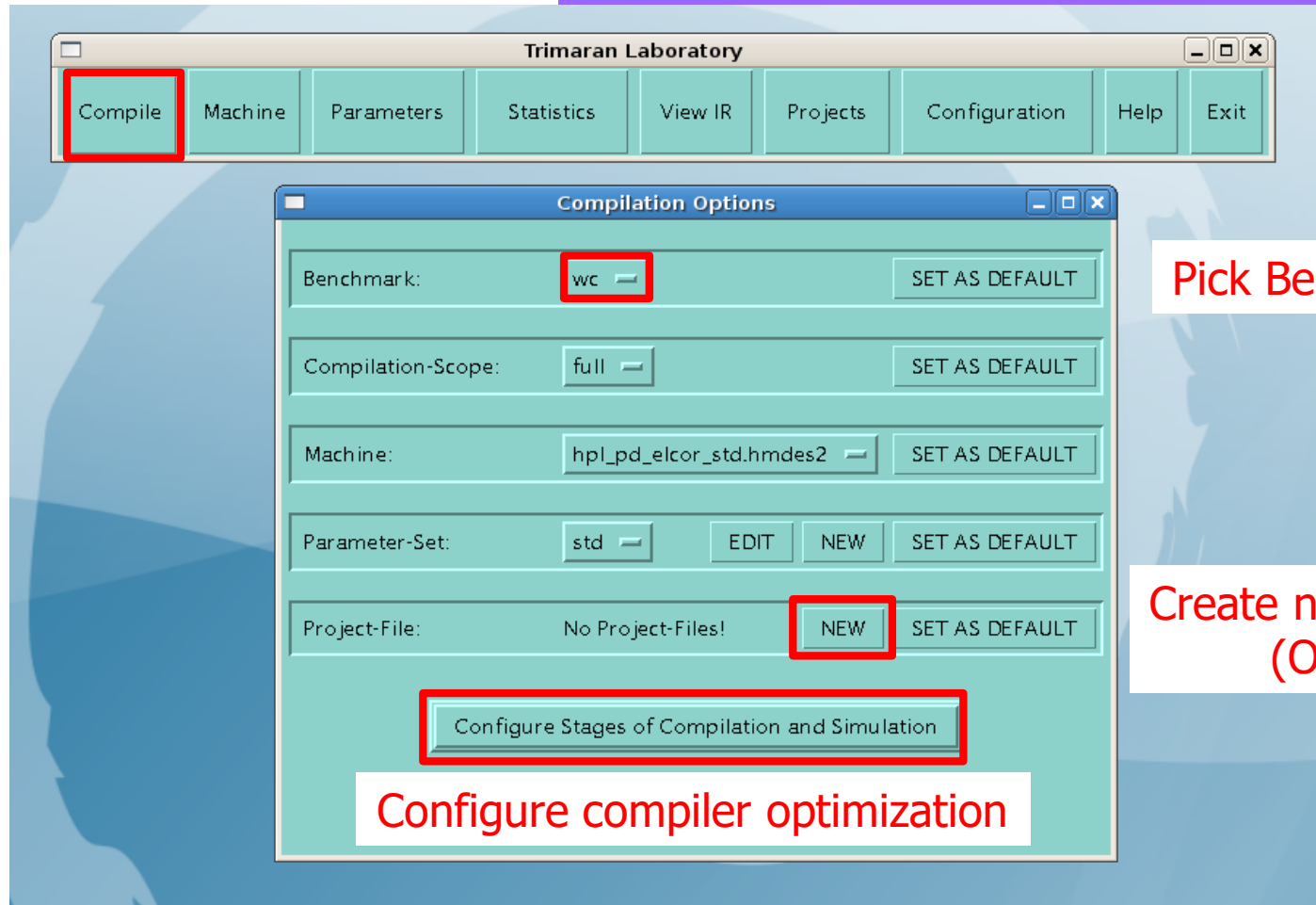
```
~/trimaran-workspace/
```



Compilation Optimization

- Generate, at compile time, several types of blocks for ILP extraction
 - ❖ Basic Block Formation (.b)
 - Preserve “Basic Blocks” contents: **no unrolling**
 - ❖ Super Block Formation (.s)
 - Creates blocks with one entrance and several exits, according to the profile of the program execution
 - ❖ Hyper Block Formation (.h)
 - Creates blocks with one entrance and several exits in order to use speculating techniques: **predication**

Trimaran Compilation

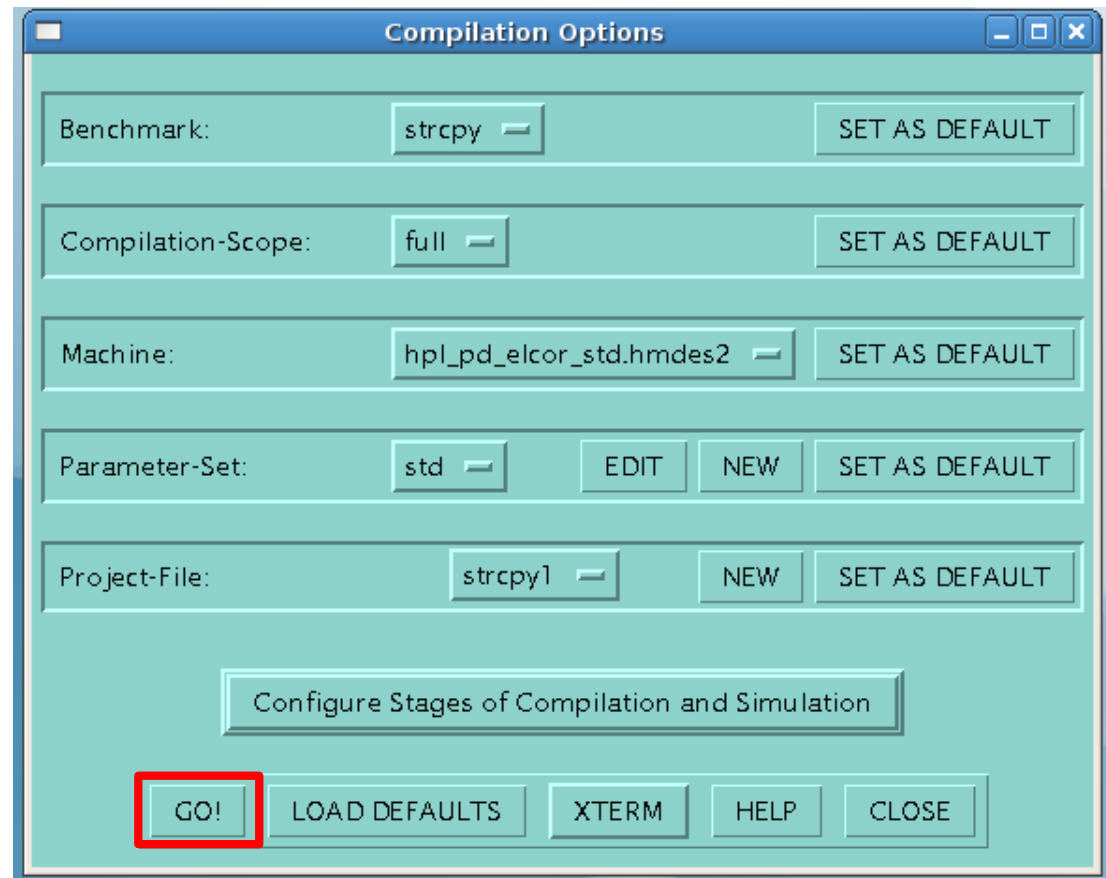
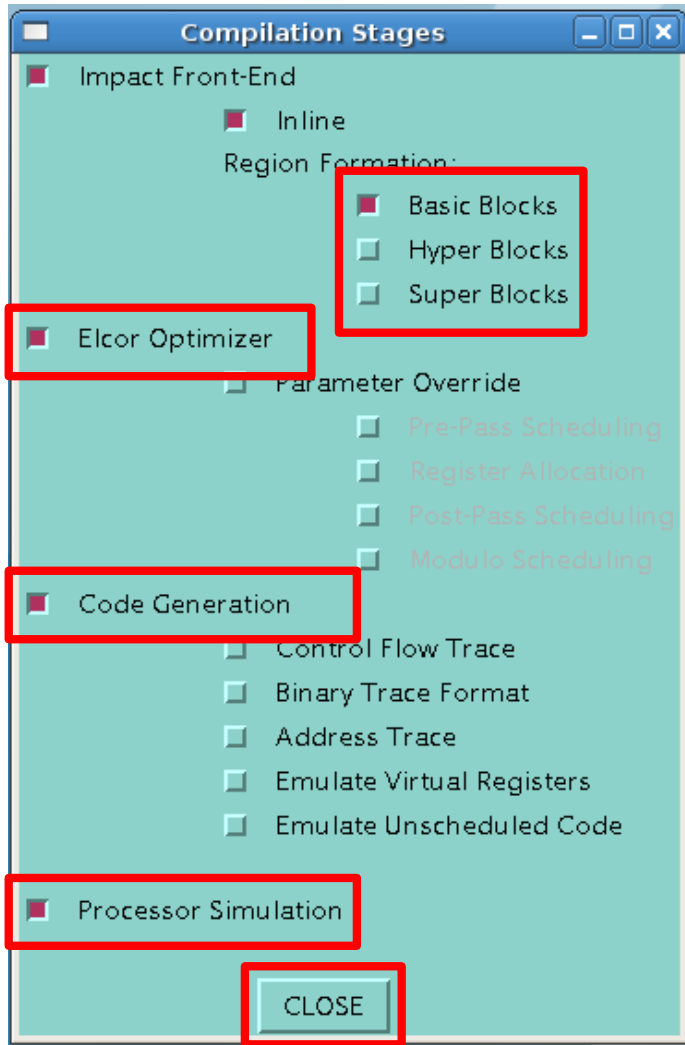


Pick Benchmark

Create new project
(Once)

Configure compiler optimization

Compiler Optimization



Compile and simulate with the GO! button

Results visualization

- ❑ Trimaran allows for 3 options to visualize compiled code
 - ❖ Program Dependency Graph
 - ❖ Control Flow Graph
 - ❖ Cycle-by-Cycle Schedule
- ❑ Visualization helps us understand the effect of compilation optimizations on performance

Government	Percentage
Current government	85%
Previous government	15%



Color Coding

Program Dependency Graph

- The different dependencies are represented by colors
 - ❖ Red: RAW
 - ❖ Lilac: WAR
 - ❖ Green: WAW
- Numbers bound with dependencies represent the corresponding latencies

Color Coding

Cycle-By-Cycle Schedule

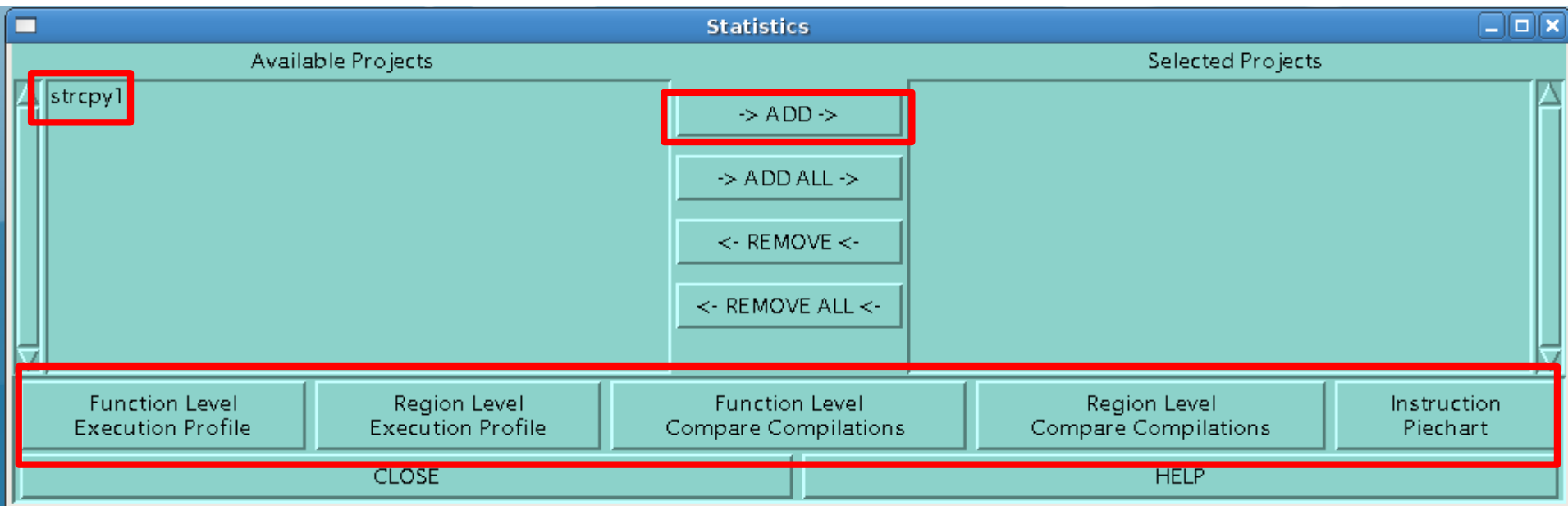
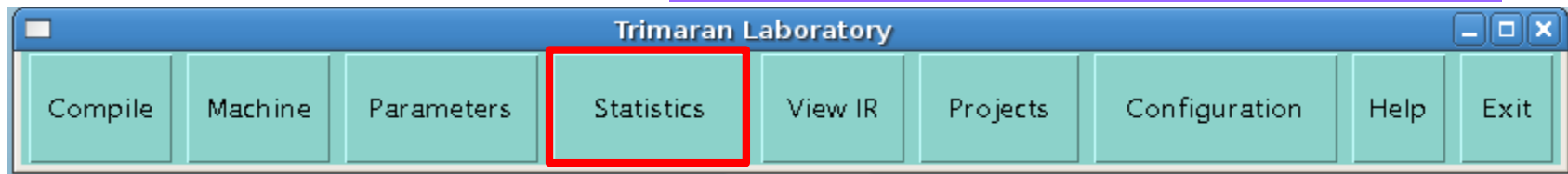
□ The different instruction types are represented by the following colors

- ❖ Red: ALU
- ❖ Magenta: Prepare-to-Branch (PBRR, ...)
- ❖ Lilac: Branch-on-Condition (BRCT, ...)
- ❖ Green: Compare-to-predicate (CMPP, ...)
- ❖ Blue: load/store

Statistics

- ❑ VLIW code generated and its performance can be quantified using different parameters
 - ❖ Number of cycles executed
 - ❖ Number of operations executed
 - ❖ $IPC = \text{Number of operations} / \text{Number of cycles}$
 - ❖ Code size generated
- ❑ Trimaran provides these numbers for program execution, per program region, and per instruction type

Statistics Visualization



Note: pie chart feature of trimaran has a "small" bug

ISA Particularities

❑ PBRR: Prepare-to-Branch

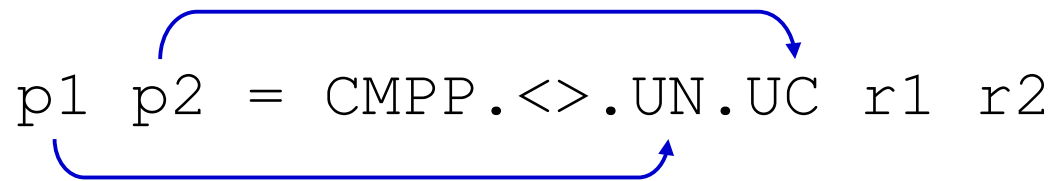
`b1 = PBRR Loop 1`

- ❖ Saves the “loop address” in b1 which is a “Branch Target Register”
- ❖ The second operand indicates the static prediction of the branch (1=taken)
- ❖ This instruction is always requested before a branch

ISA Particularities

❑ CMPP: Compare-to-predicate

$p1 \quad p2 = \text{CMPP}.\langle \rangle.\text{UN}.\text{UC} \quad r1 \quad r2$



- ❖ Compares the r1 and r2 registers and fixes the p1 and p2 predicates
- ❖ UN: Unconditional Normal
- ❖ UC: Unconditional Complement

ISA Particularities

❑ BRCT: Branch-on-Condition

BRCT b1 p1

- ❖ Jumps to the address contained in b1 (Branch Target Register) if the p1 predicate is true

Loop Unrolling

- ❑ We explore loop unrolling with strcpy benchmark
- ❑ Benchmark source code

`~/trimaran-workspace/benchmarks/strcpy/src/`

Benchmark Source Code

```
#strcpy.c
#include <stdio.h>
#define BUFLLEN 1024
char A[BUFLLEN], B[BUFLLEN];
main(){
char *a, *b;
int i;
// init null-terminated string in
buf A
for (i=0; i<BUFLLEN-1; i++)
A[i] = 'a'+(i%26);
A[i] = '\\0';
/* copy string in A to buffer B */
a = &A[0];
b = &B[0];
while (*a != 0)
*b++ = *a++;
}
```

No Unrolling Compilation

❑ Compile strcpy with Basic Block Formation (slides 8-9)

- ❖ Open "Compile"
- ❖ Benchmark "strcpy"
- ❖ Parameter-Set "std"
- ❖ Project-File "NEW" "strcpy"
- ❖ Compilation and Simulation
 - Basic Blocks
 - Code Generation
 - Processor Simulation

❖ GO!

No Unrolling Statistics

□ From the statistics (slide 15):

- ❖ What is the number of execution cycles?
- ❖ What is the number of execution operations?
- ❖ What is the IPC?
- ❖ What is the size of the code?

No Unrolling Assembly

- ❑ The source code has the following loop (slide 20)

```
while (*a != 0)
    *b++ = *a++;
```

- ❑ Observe the “Control Flow” representation of the loop and explain the generated assembly (next slide)

Note: In Trimaran, the address registers (b1) are represented with the same name as the GP registers. Consider the semantic of the instruction to know whether r1 designates an address register or a general purpose register

- ❑ Why are there two loads inside the loop?

No Unrolling Assembly

```
r2 = _B
```

```
r7 = _A - _B
```

```
r8 = r7 + 1
```

BB5:

```
r5 = r2 + r7
```

```
r6 = r2 + r8
```

```
r2 = pbr r BB5 1
```

```
r3 = ld r5
```

```
r3 = exts_b r3
```

```
st r2 r3
```

```
r4 = ld r6
```

```
r2 = r2 + 1
```

```
r4 = exts_b r4
```

```
p2 u = cmpp.<>.UN.UN r4 0
```

```
brct r2 p2
```

Warning

Trimaran appears to give general purpose and branch target register (GPR and BTR, respectively) the same names: *r2* in this example, as in *r2 = r2 + 1* and *r2 = pbr r BB5 1*.

Although it is most likely a bug, it is not of practical consequence, for the destination of *pbr r* or the source of *brct* can only be a BTR, so that *r2* is clearly another *r2* than the source and destination of the addition.

Loop Unrolling

- ❑ Run Super Block Formation (slide 21)
- ❑ Observe, in the Control Flow representation, the while loop studied in previous slide
- ❑ What changes can you notice? How many times is the loop executed?
- ❑ Compare the statistics of the two executions

Loop Unrolling Assembly

- ❑ Compare the Cycle-By-Cycle Schedule with the previous execution and explain the improvements made in the new assembly code
- ❑ Why is there a branch at every iteration in loop body?
- ❑ Compare the two loops of the strcpy program (for and while), what are the major differences?

Loop Unrolling Phase

- Describe the phases of block execution: Head, Body, & Tail

