



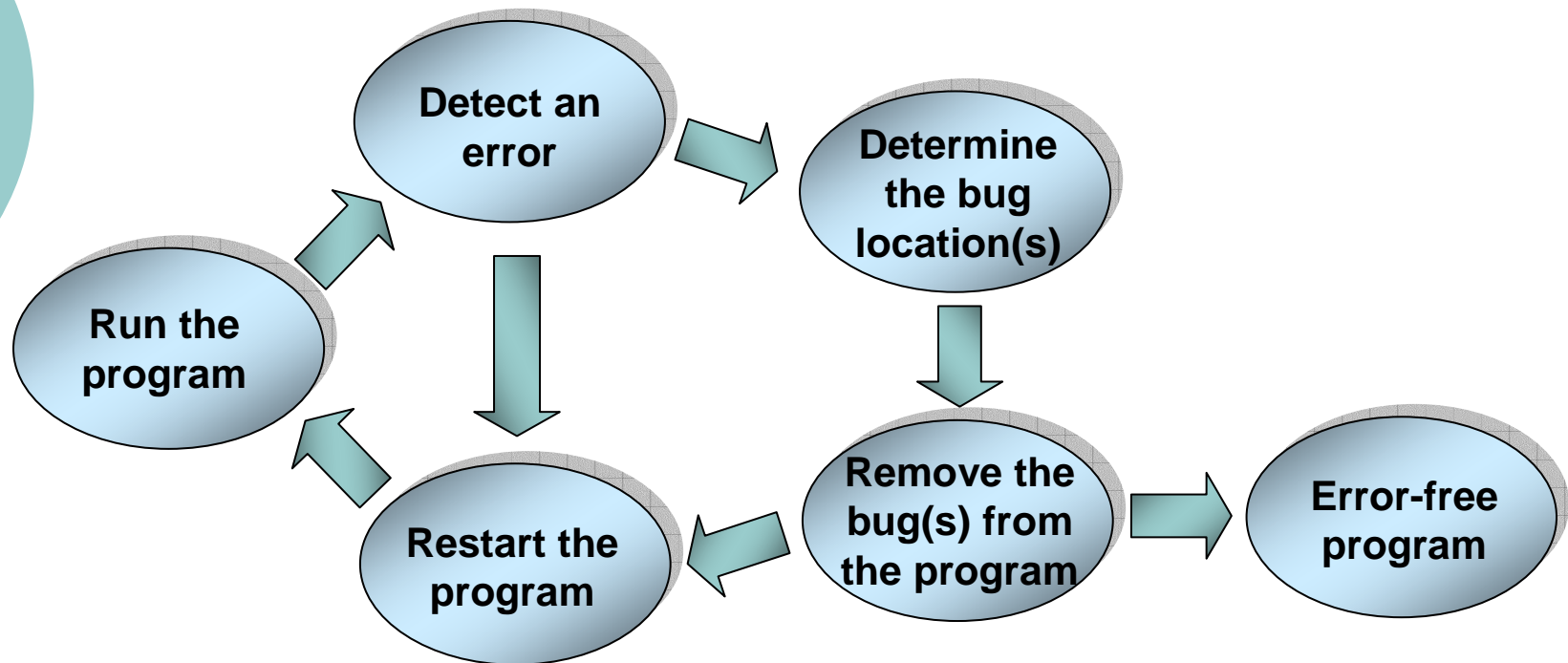
Instruction-level Reverse Execution for Debugging

Tankut Akgul and Vincent J. Mooney

School of Electrical and Computer Engineering
Georgia Institute of Technology

November 2002

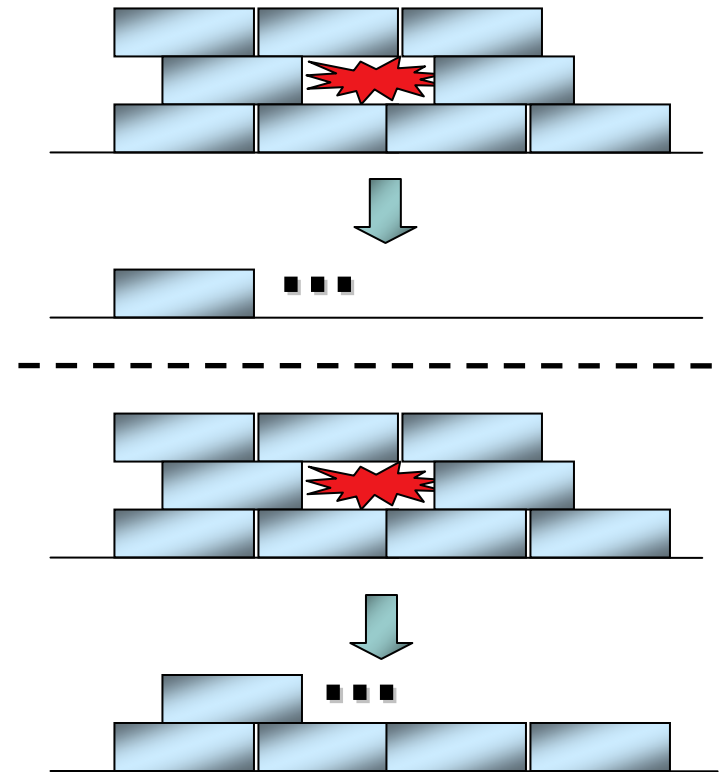
Background



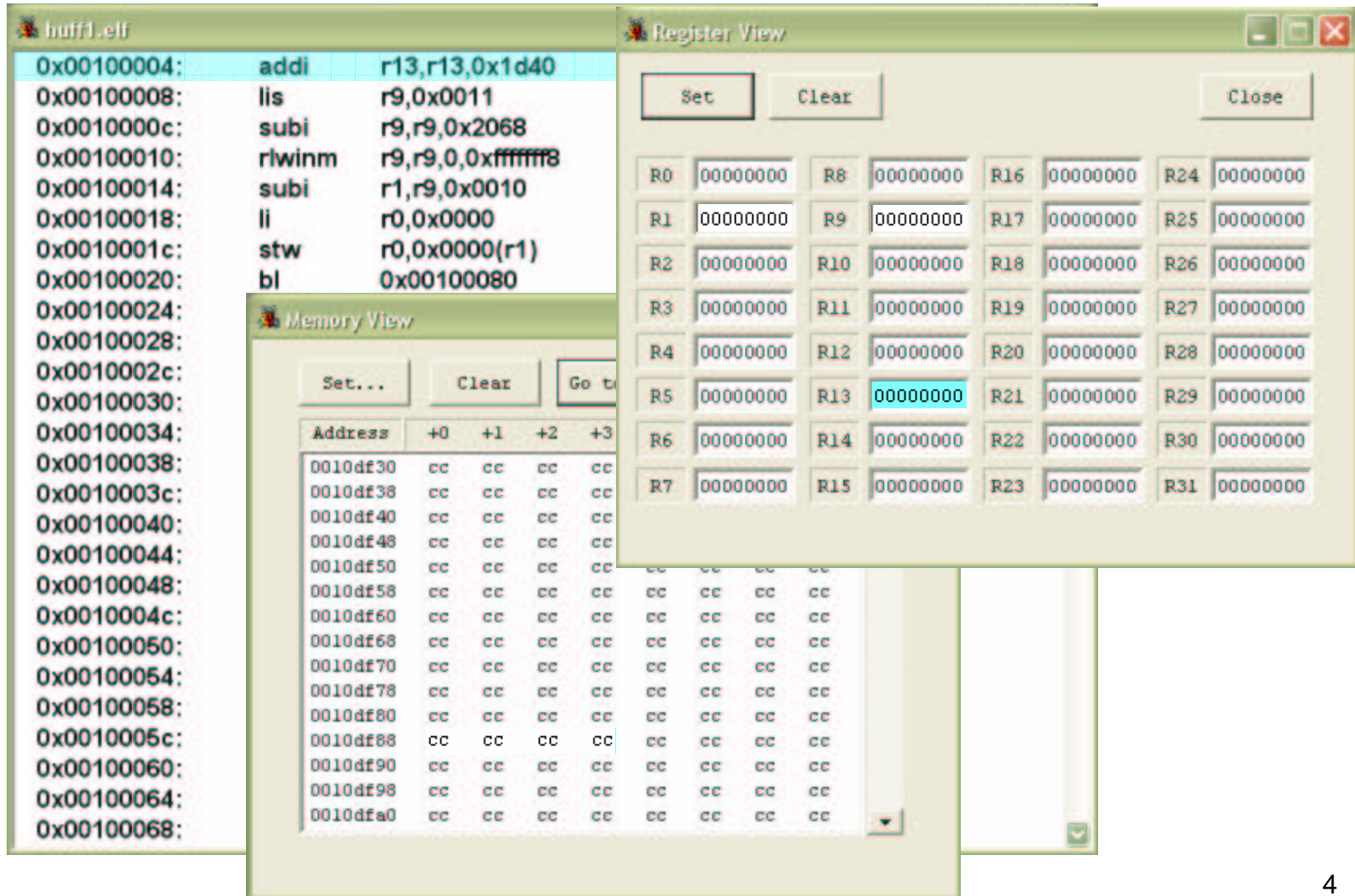
Debugging is a repetitive process!

Motivation

- State constructed during forward execution
- At least one (typically more than one) re-execution required for locating a bug in a program
- Re-executions localized around erroneous program points by reverse execution
- Time saved by preventing re-executions starting from the beginning of the program
- Assembly-level reverse execution as a first step



Motivation



The screenshot displays a debugger interface with three main windows:

- Assembly View:** Shows assembly instructions for a file named 'huff1.elf'. The instructions are:
 - 0x00100004: `addi r13,r13,0x1d40`
 - 0x00100008: `lis r9,0x0011`
 - 0x0010000c: `subi r9,r9,0x2068`
 - 0x00100010: `rlwinm r9,r9,0,0xfffff8`
 - 0x00100014: `subi r1,r9,0x0010`
 - 0x00100018: `li r0,0x0000`
 - 0x0010001c: `stw r0,0x0000(r1)`
 - 0x00100020: `bl 0x00100080`
- Register View:** Shows the state of 32 registers (R0 to R31). Register R13 is highlighted with a blue selection box and contains the value 00000000.
- Memory View:** Shows a memory dump starting at address 0010df30. The dump consists of a grid of hexadecimal values, all of which are 'cc'.

Previous Work

- Periodic state saving
 - Save whole processor state periodically
- Incremental state saving
 - Save modified processor state
- Program animation
 - Construct a virtual machine with reversible instructions which are usually stack operations
- Source transformation
 - Transform the source code to a reversible source code version
 - Apply state saving for destructive statements

All above methods use state saving heavily!

State saving = **time** and **memory** overheads introduced during forward execution

Methodology

We define the state of a processor as follows:

$$S = (PC, M, R)$$

PC : program counter

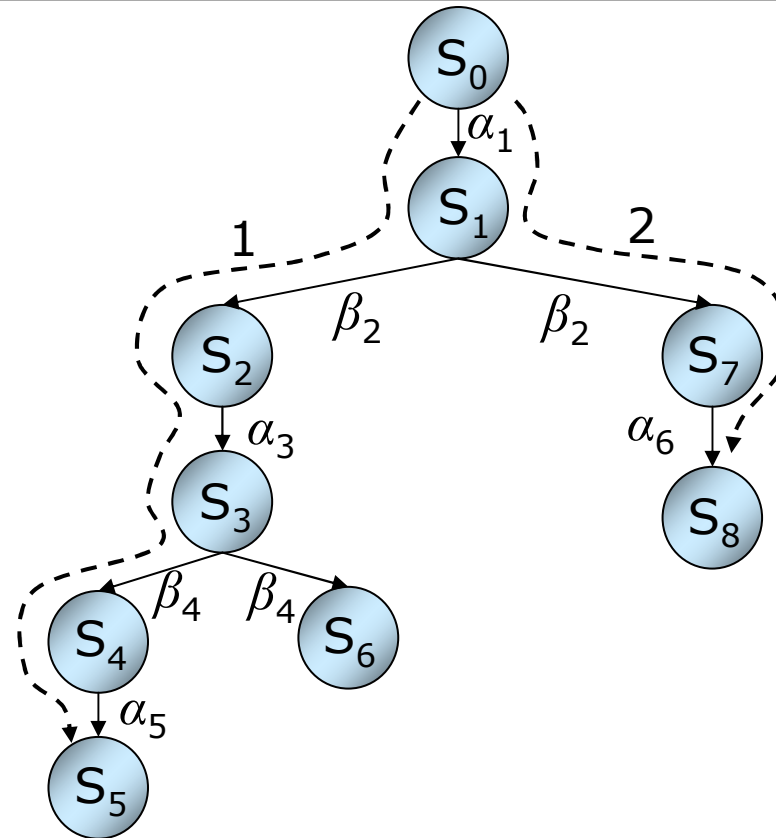
M : memory values

R : register values

In order to reverse execute a program do the following:

- Construct a reverse program T' for an input program T
- Recover M and R by executing T' in place of T
- Recover the program counter value with the help of the debugger tool

Program Execution Model



Execution 1: $I_1 = (\alpha_1, \beta_2, \alpha_3, \beta_4, \alpha_5)$

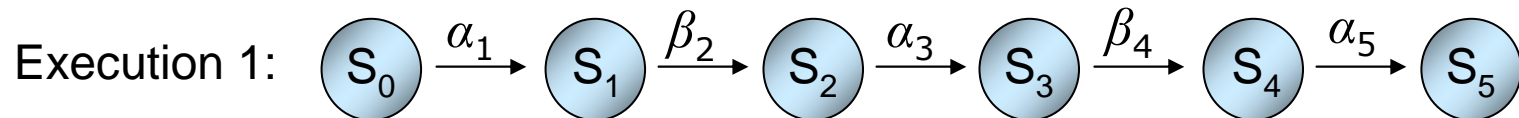
α : a non-branch instruction

Execution 2: $I_2 = (\alpha_1, \beta_2, \alpha_6)$

β : a branch instruction

Reverse Execution

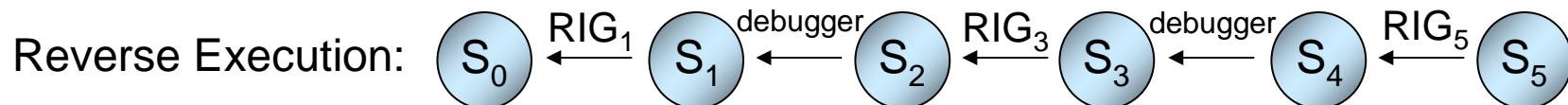
Take a specific execution of T



Generate a set of one or more reverse instructions, a *reverse instruction group (RIG)*, for every non-branch instruction such that RIG_x reverses the effect of α_x

$$(\alpha_1, \alpha_3, \alpha_5) \rightarrow (RIG_1, RIG_3, RIG_5)$$

Execute RIGs in the order **opposite to the completion order** of the instructions during forward execution and have the debugger tool recover the rest of the state



Reverse Execution (Continued)

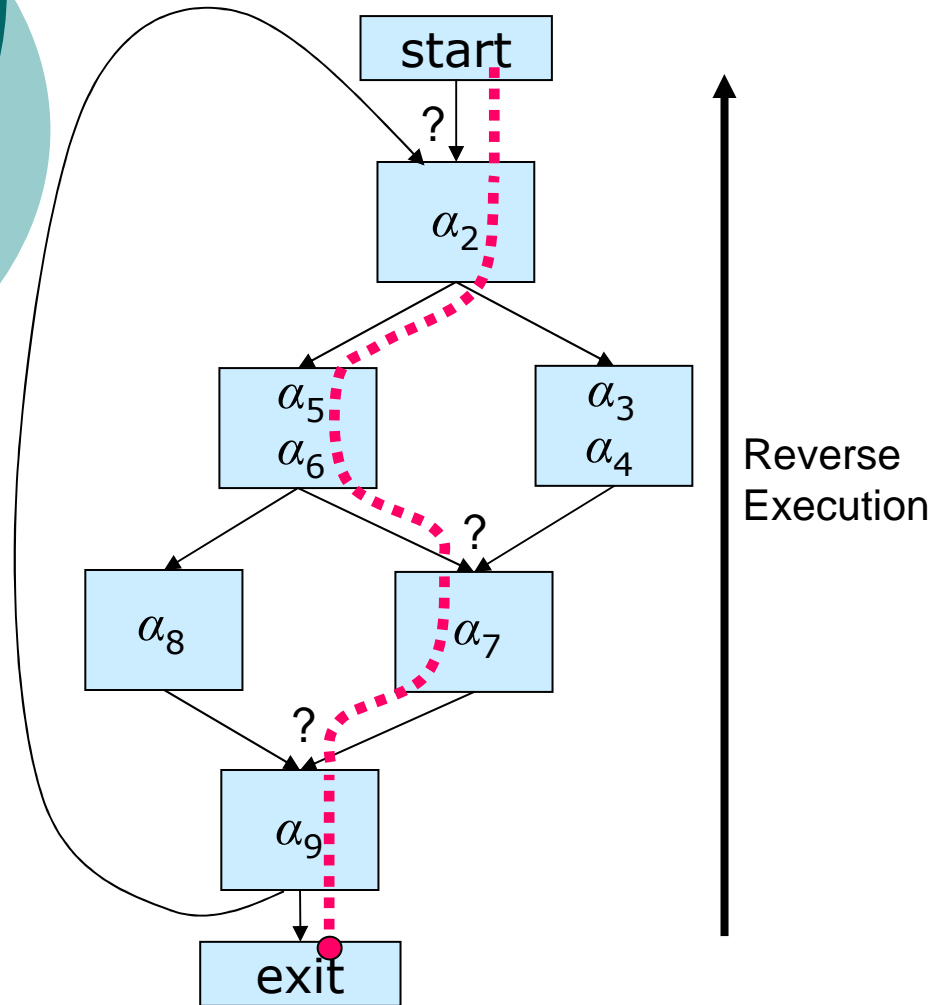
Problem:

Dynamic control flow of T may change!

Solution:

- Find out a condition set C (*predicate expressions*) which determines control flow of T
- Combine the RIGs in such a way that the execution order of the RIGs is bound to C

Reverse Execution (Continued)

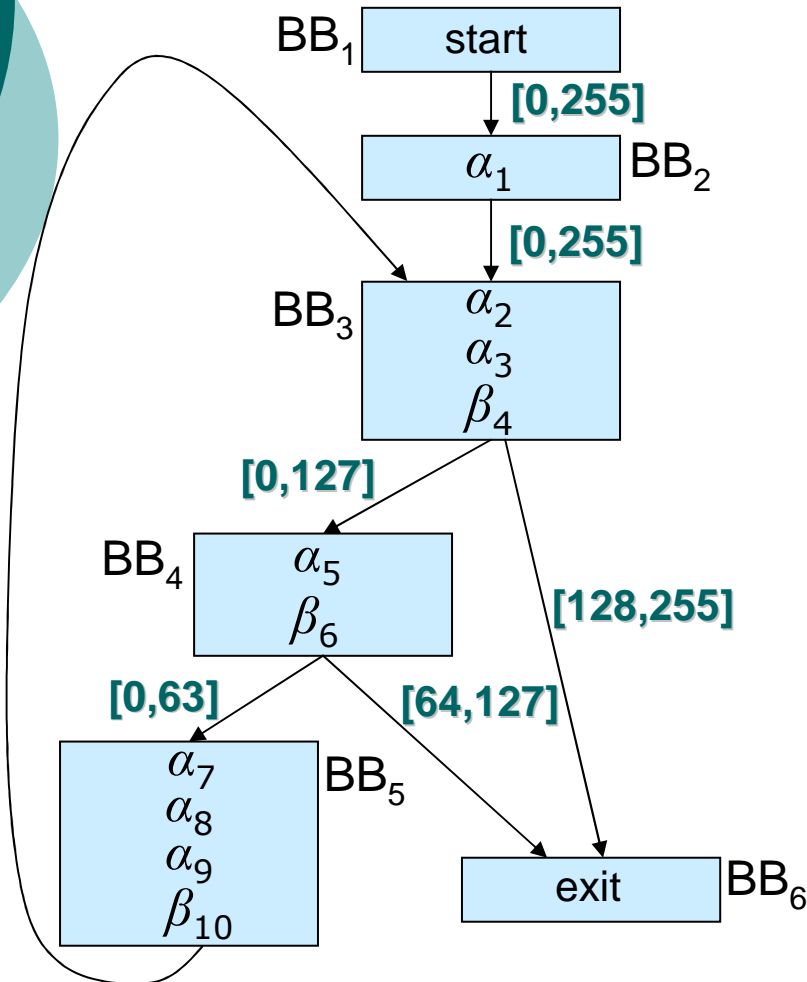


- Instructions in a basic block (BB) complete in lexical order
- Confluence points are the only decision points on the path to follow during reverse execution

Reverse Code Generation (RCG) Algorithm

- Step 1: Constructs a *control flow graph* (CFG) for every procedure/function (intra-procedural analysis) and labels the CFG edges for Step2
- Step 2: Determines the predicate expressions (condition set C) at the confluence points in the CFG of each procedure/function
- Step 3: Constructs the RIGs
- Step 4: Combines the RIGs via conditional branch instructions with the determined predicates at the confluence points to generate the reverse of each procedure/function
- Step 5: Combines the reverse procedures/functions

Step 1: CFG Construction and Labeling



$L_{i,j}^{in}$: j^{th} incoming forward edge of BB_i

$L_{i,j}^{out}$: j^{th} outgoing forward edge of BB_i

$$L_{1,1}^{out} = [0,255]$$

$\forall BB_i \in \text{CFG} - \{\text{exit} \wedge \text{start}\}, \text{ do } \{$

$$L^{\text{temp}} = \bigcup_{k=1}^n [x_k, y_k] = \bigcup_{j=1}^{|\text{InFwdEdges}(BB_i)|} L_{i,j}^{in}$$

for $k=1$ to n {

if $(|\text{OutFwdEdges}(BB_i)| == 2)$ {

$$L_{i,1}^{out} = L_{i,1}^{out} \cup [x_k, (x_k + y_k + 1)/2 - 1],$$

$$L_{i,2}^{out} = L_{i,2}^{out} \cup [(x_k + y_k + 1)/2, y_k]$$

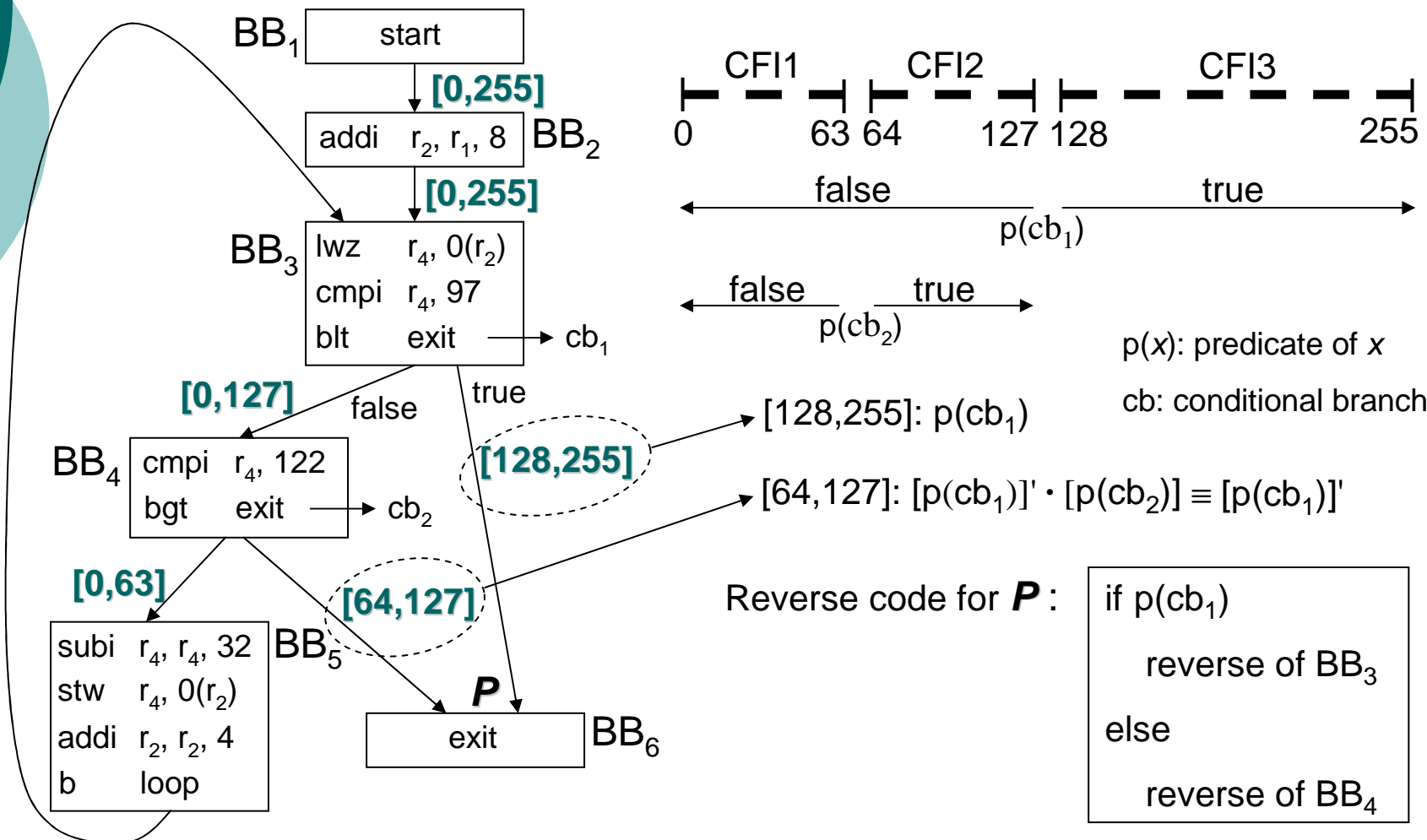
} else if $|\text{OutFwdEdges}(BB_i)| == 1$

$$L_{i,1}^{out} = L_{i,1}^{out} \cup [x_k, y_k]$$

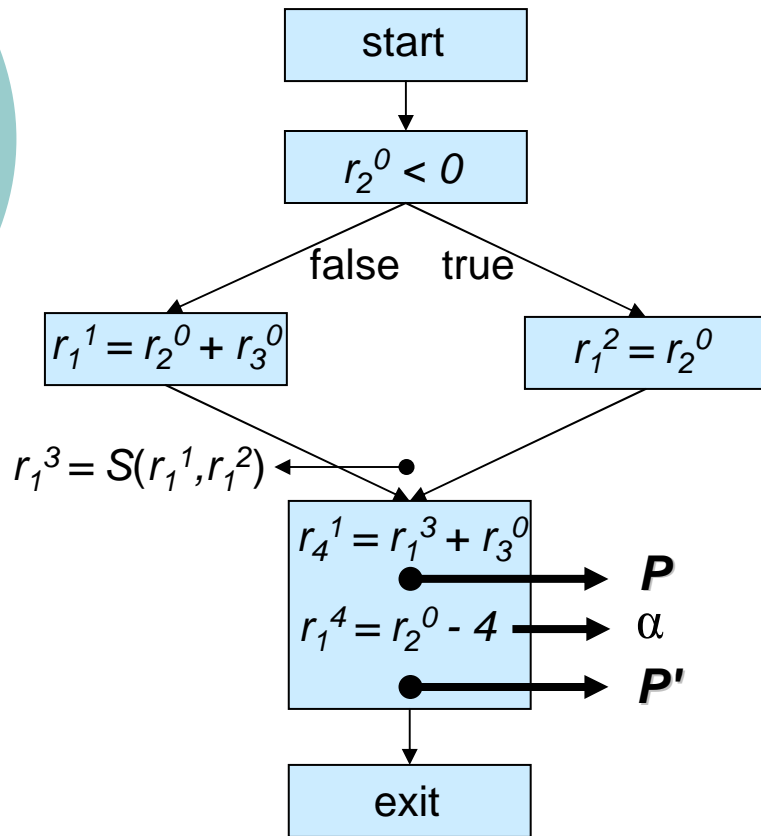
}

}

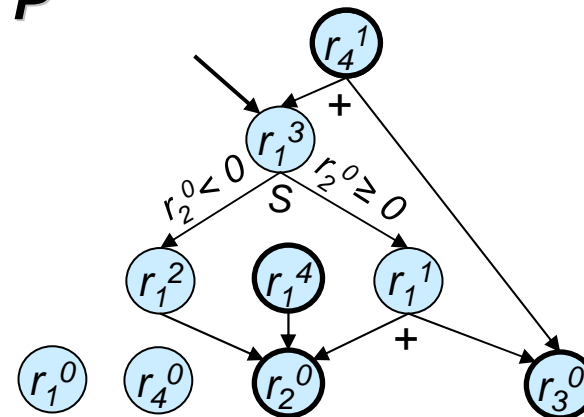
Step 2: Predicate Expression Determination



Step 3: Construction of the RIGs



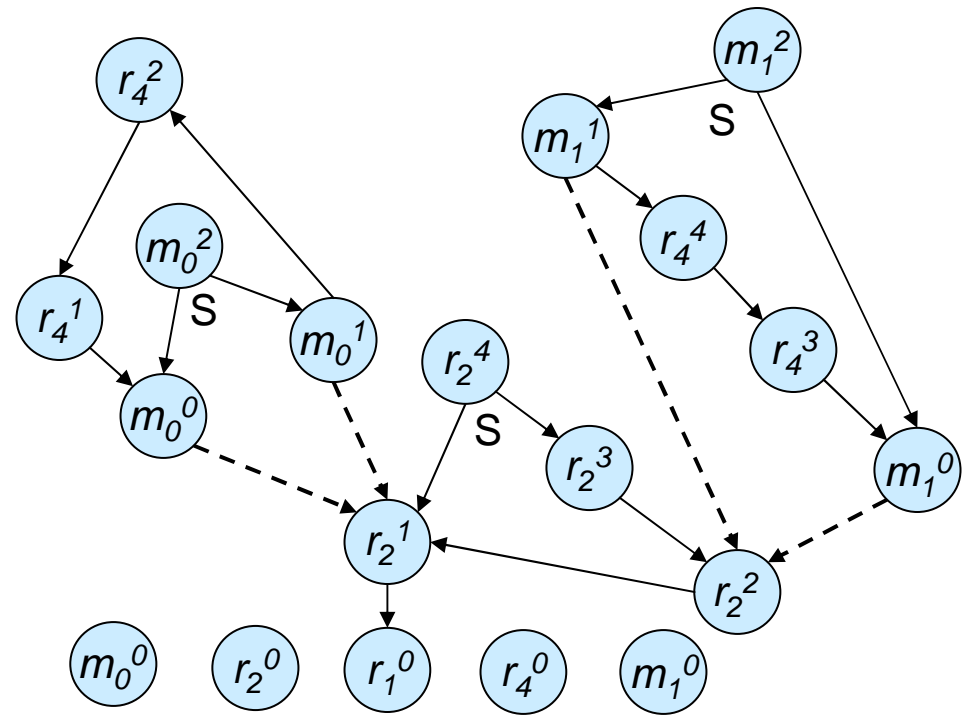
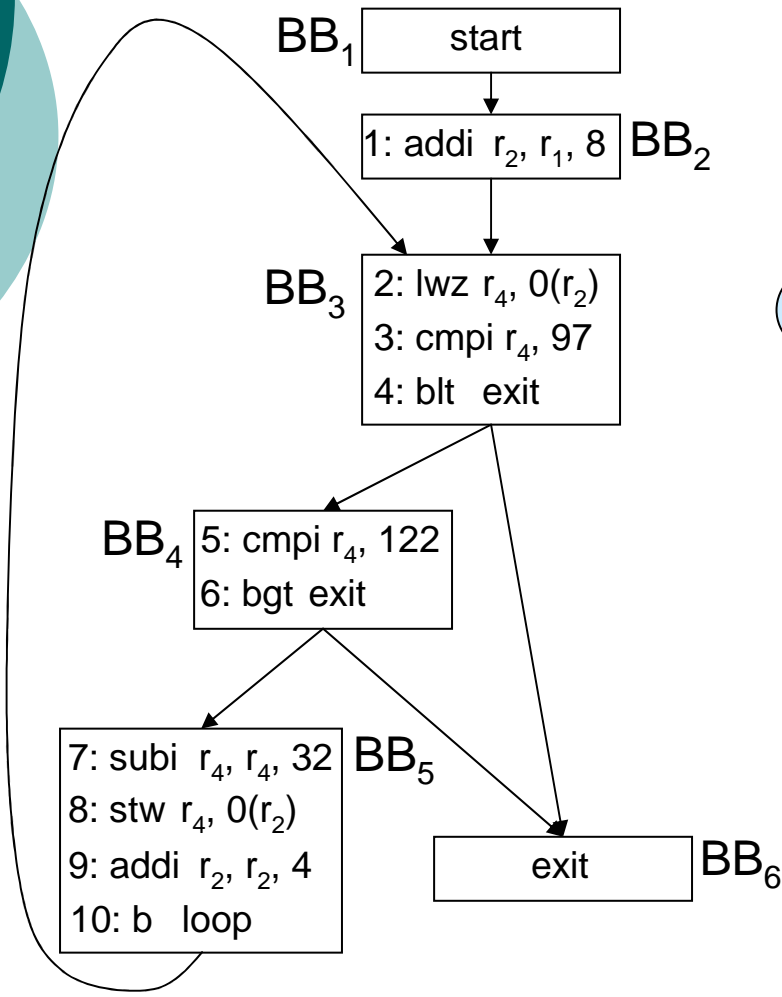
- Rename Values
- Generate a *directed acyclic graph (DAG)*
- Find the definition of r_1 reaching P
- Recover r_1 using available nodes at P'



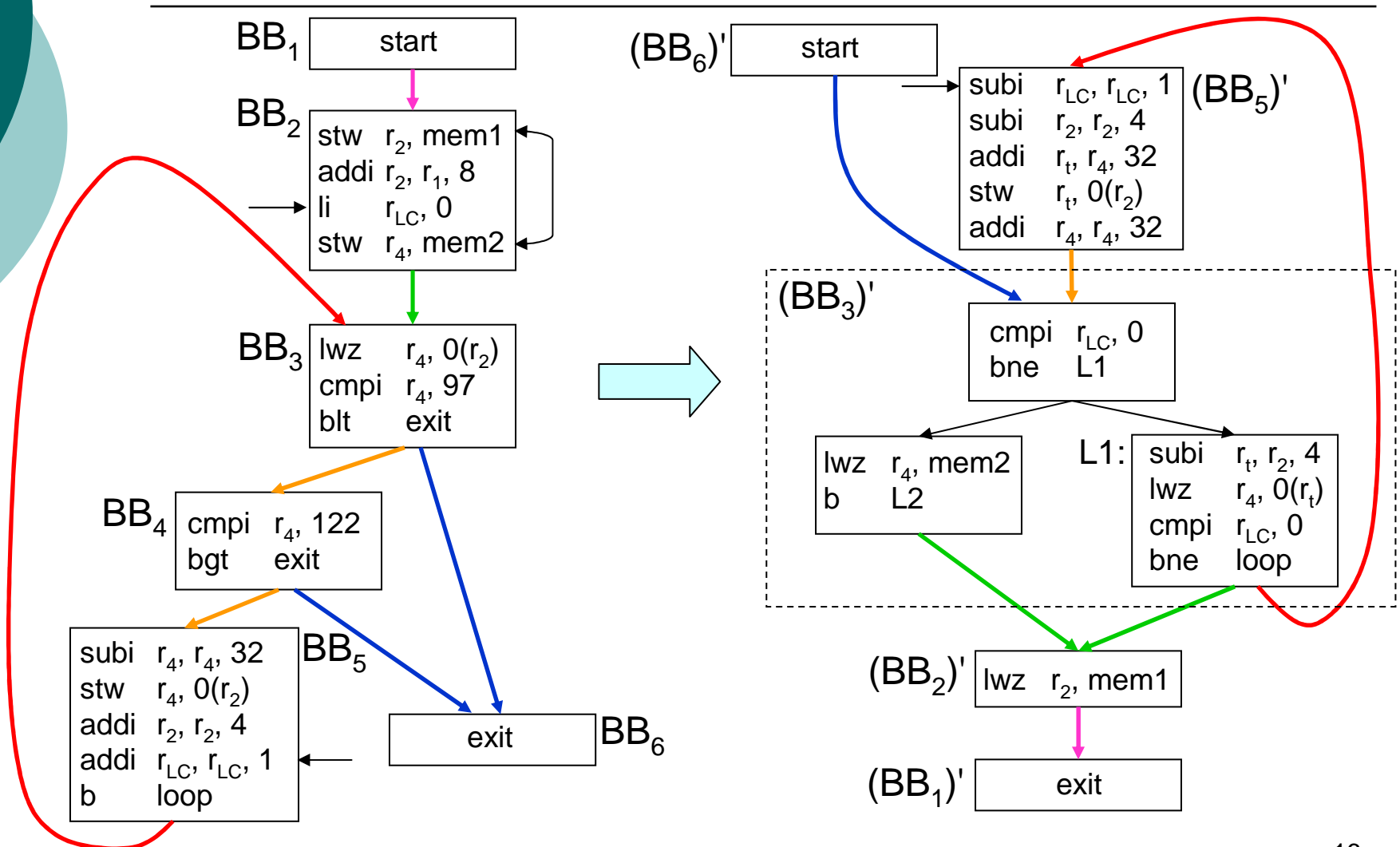
RIG for α : $r_1 = r_4 - r_3$ or

$\begin{aligned} &\text{if } r_2 < 0 \\ & \quad r_1 = r_2 \\ & \text{else} \\ & \quad r_1 = r_2 + r_3 \end{aligned}$
--

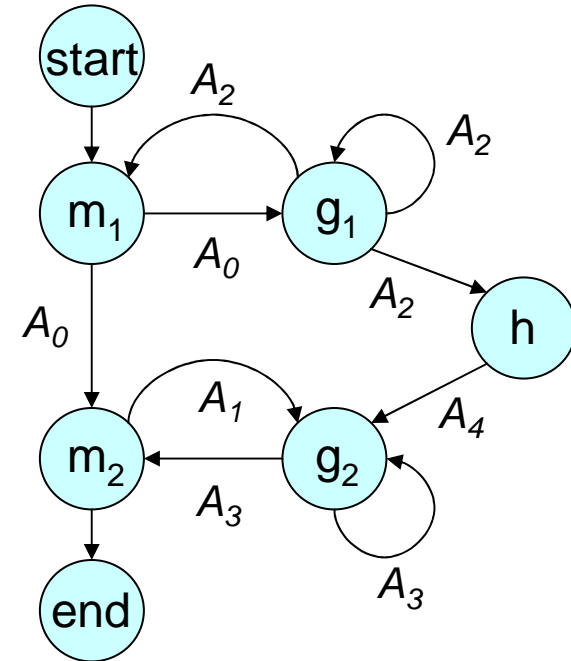
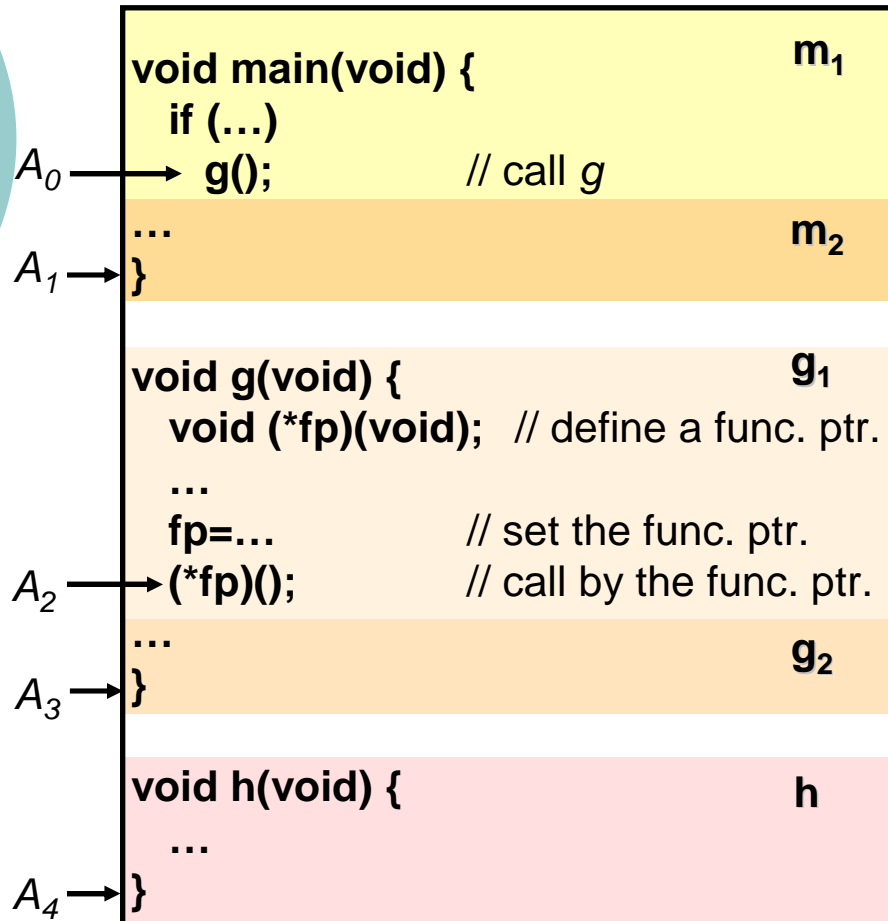
Step 3: Construction of the RIGs (Example)



Step 4: Combination of the RIGs

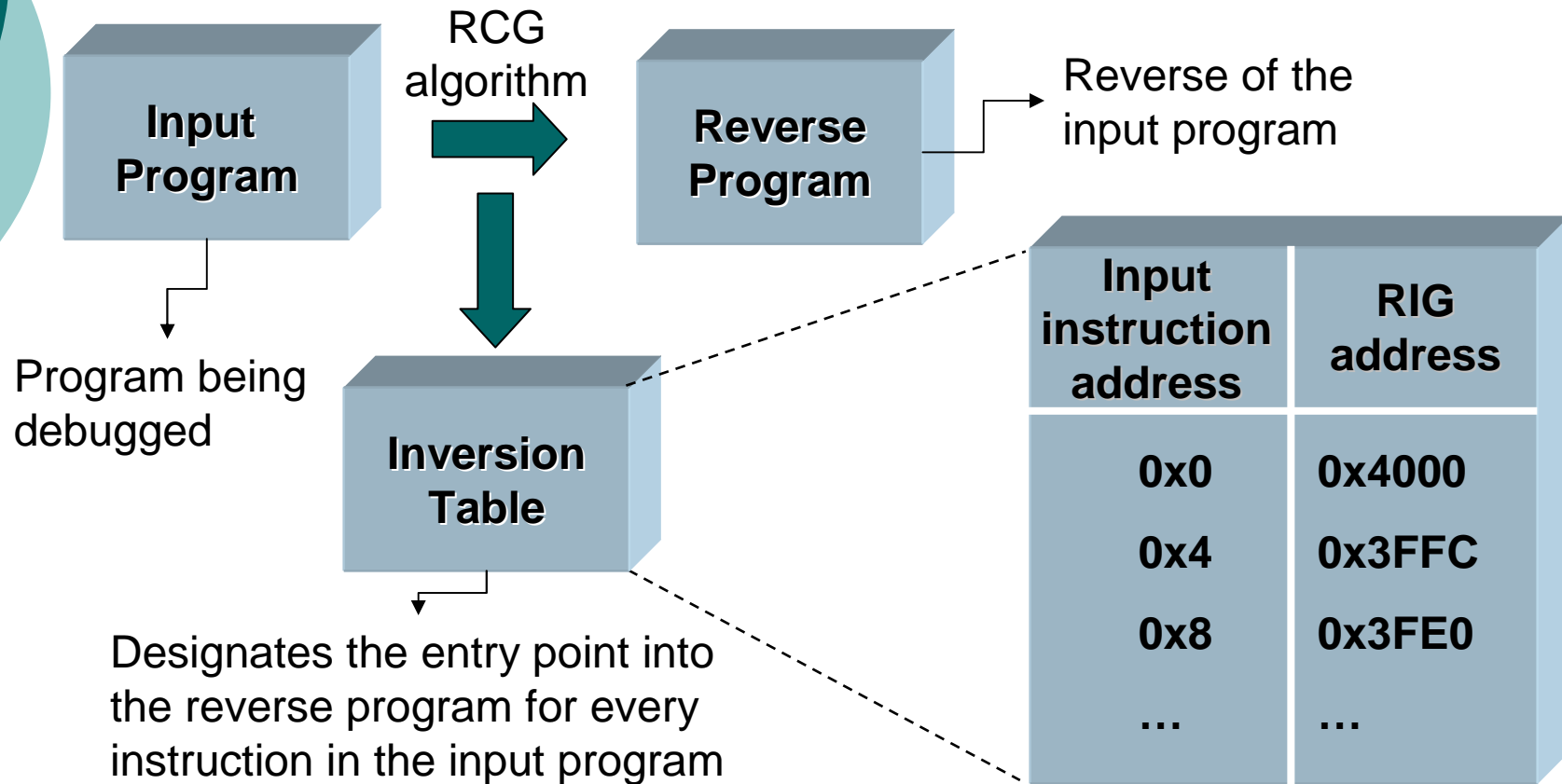


Step 5: Combination of Reverse Procedures/functions



- Push addresses on the dynamically taken edges into stack
- Pop the addresses from stack during reverse execution and branch to popped addresses

Recovering the Program Counter



Experimentation Platform

Host

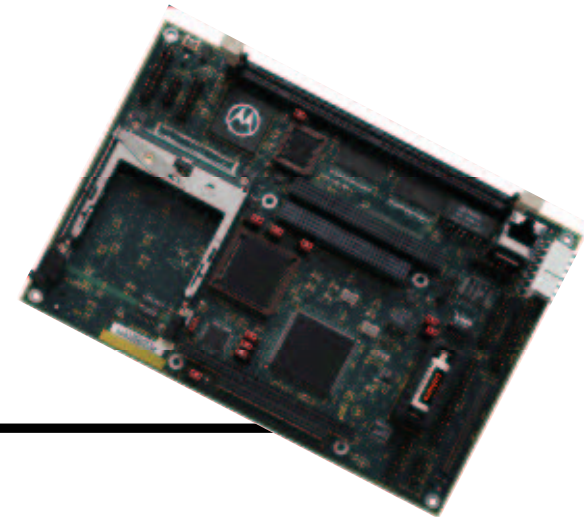
Target



PC

Windows 2000

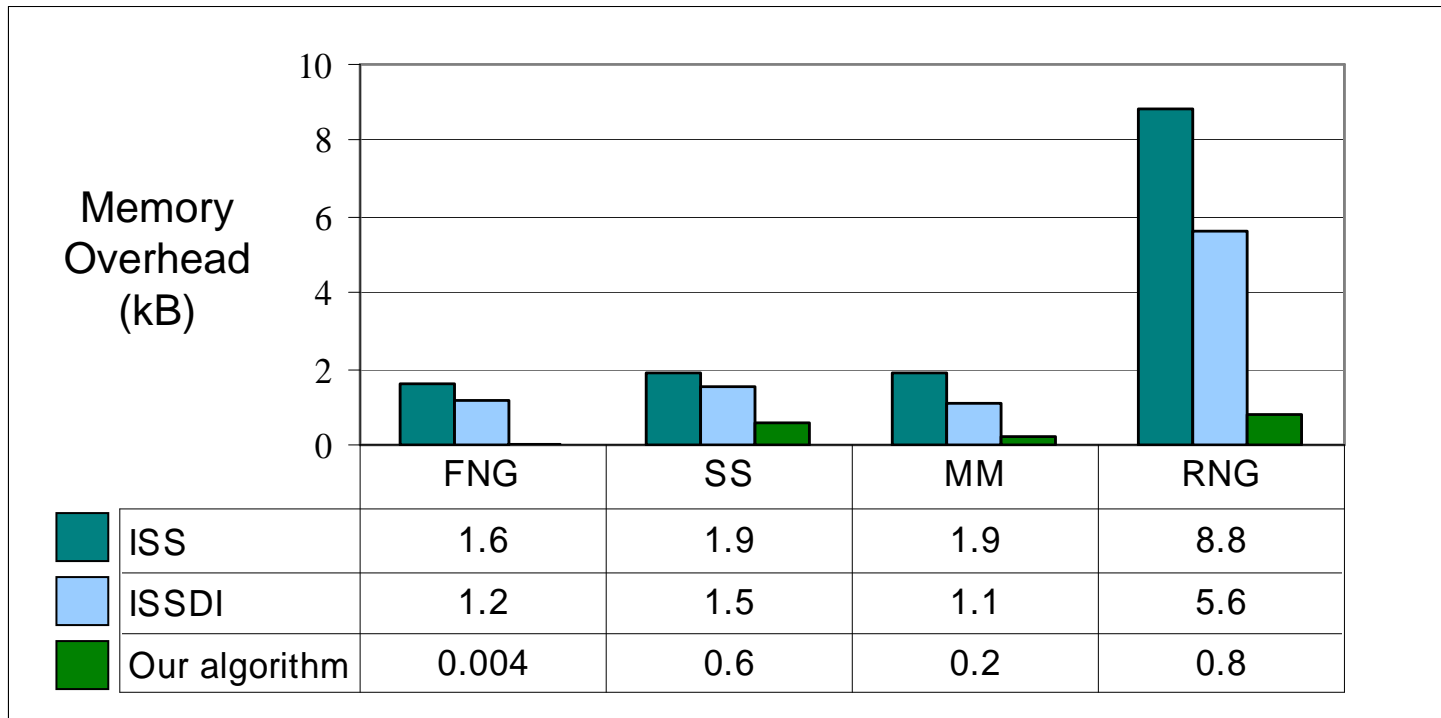
Background Debug
Mode (BDM) Interface



MBX860

MPC860 processor
4MB DRAM, 2MB Flash
RTC, four 16-bit timers, watchdog

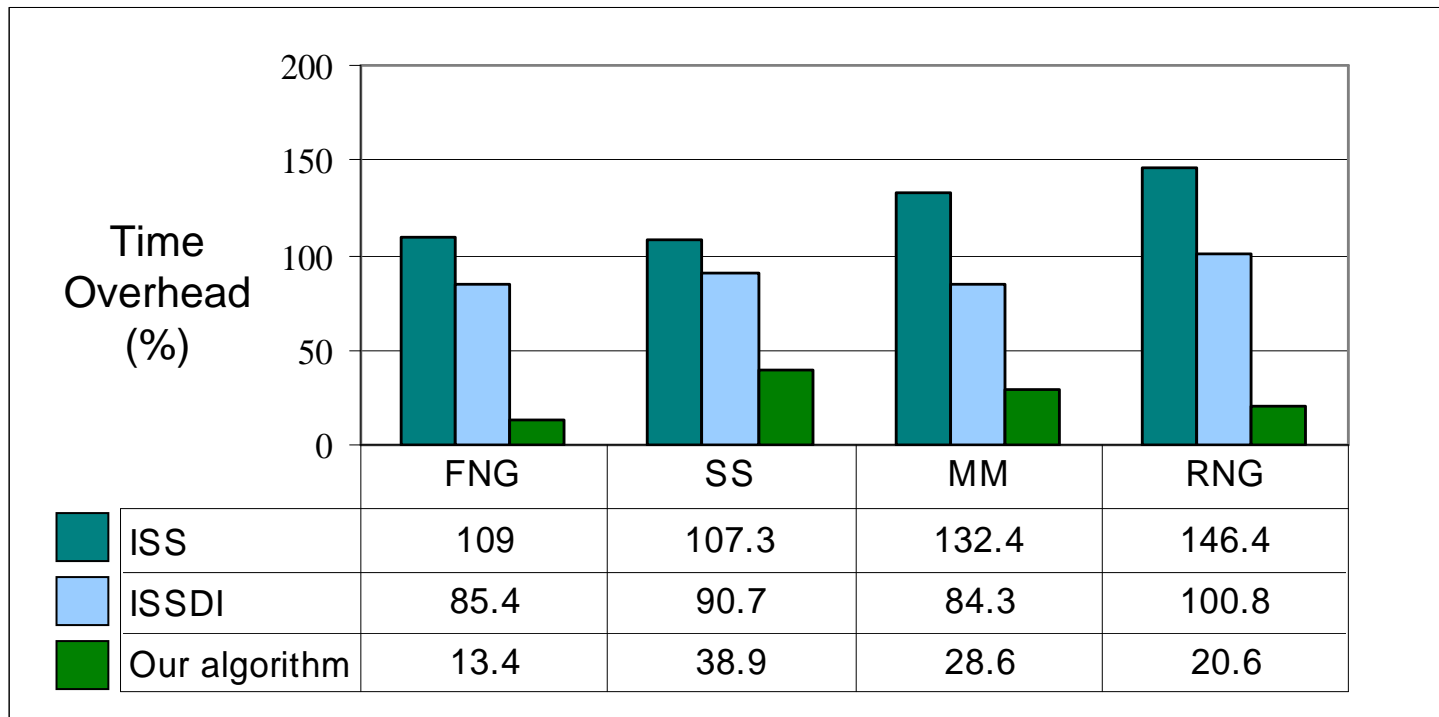
Experimental Results



ISS: Incremental State Saving, ISSDI: Incremental State Saving for Destructive Instructions

FNG: Fibonacci Number Generator, SS: Selection Sort, MM: Matrix Multiply, RNG: Random Number Generator

Experimental Results (Cont.)



ISS: Incremental State Saving, ISSDI: Incremental State Saving for Destructive Instructions

FNG: Fibonacci Number Generator, SS: Selection Sort, MM: Matrix Multiply, RNG: Random Number Generator

Reverse Debugger

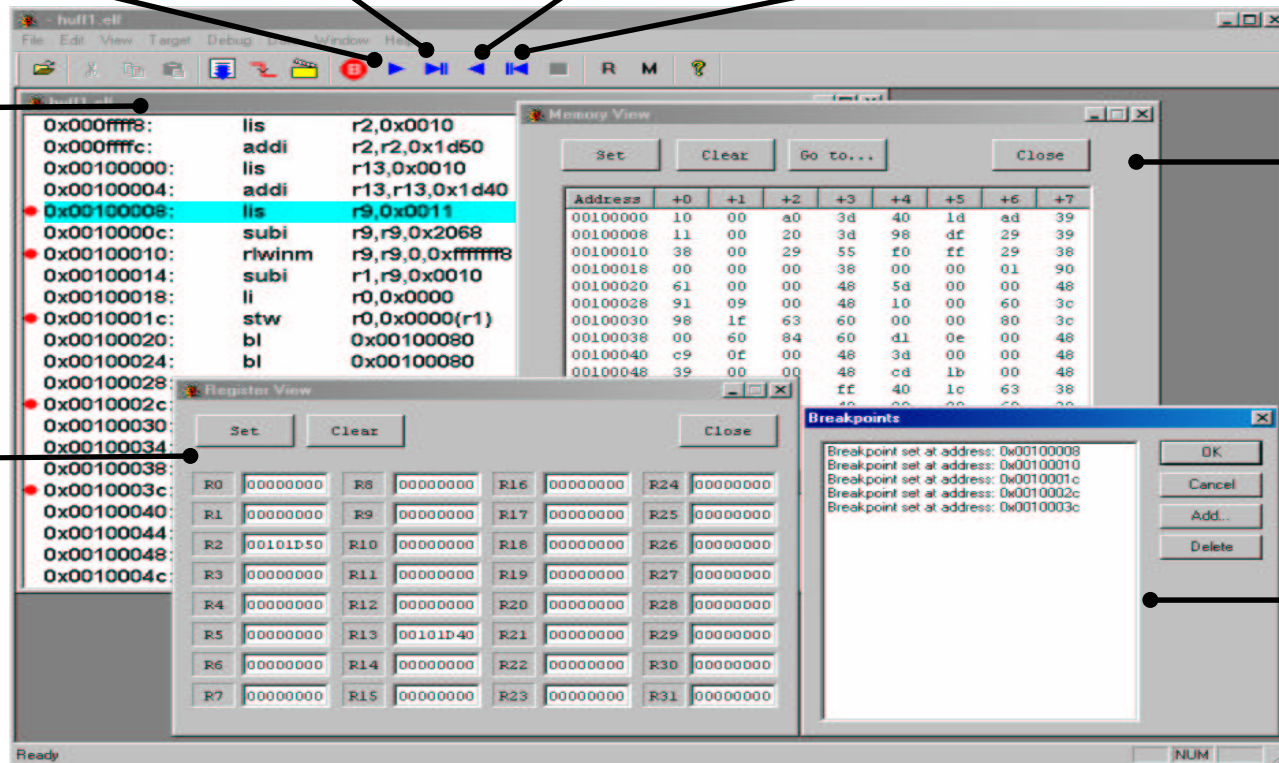
Execute forward Step forward Execute backward Step backward

Source window

Memory window

Register window

Breakpoint window



The screenshot shows a reverse debugger interface with several windows and a dialog box:

- Source window:** Displays assembly code with addresses and instructions. The instruction at address 0x00100008 is highlighted: `lis r9,0x0011`.
- Memory window:** Shows a table of memory addresses and their corresponding values.
- Register window:** Shows a grid of registers (R0-R31) and their current values.
- Breakpoint window:** A dialog box listing breakpoints set at addresses: 0x00100008, 0x00100010, 0x0010001c, 0x0010002c, and 0x0010003c.
- Toolbar:** Contains navigation buttons for 'Execute forward', 'Step forward', 'Execute backward', and 'Step backward'.

Conclusion

- Reduced debugging time with localized re-executions
- Very low time and memory overheads in forward execution by using reverse code
- Reverse execution up to an assembly instruction level granularity

T. Akgul and V. J. Mooney. Instruction-level reverse execution for debugging. Technical Report GIT-CC-02-49, Georgia Institute of Technology, September 2002.

http://www.cc.gatech.edu/tech_reports/index.02.html