Karlsruhe Institute of Technology
Institut für Technische Informatik
Chair for Embedded Systems – Prof. Dr. J. Henkel

Vorlesung im SS 2016

Reconfigurable and Adaptive Systems (RAS)

Marvin Damschen, Lars Bauer, Jörg Henkel

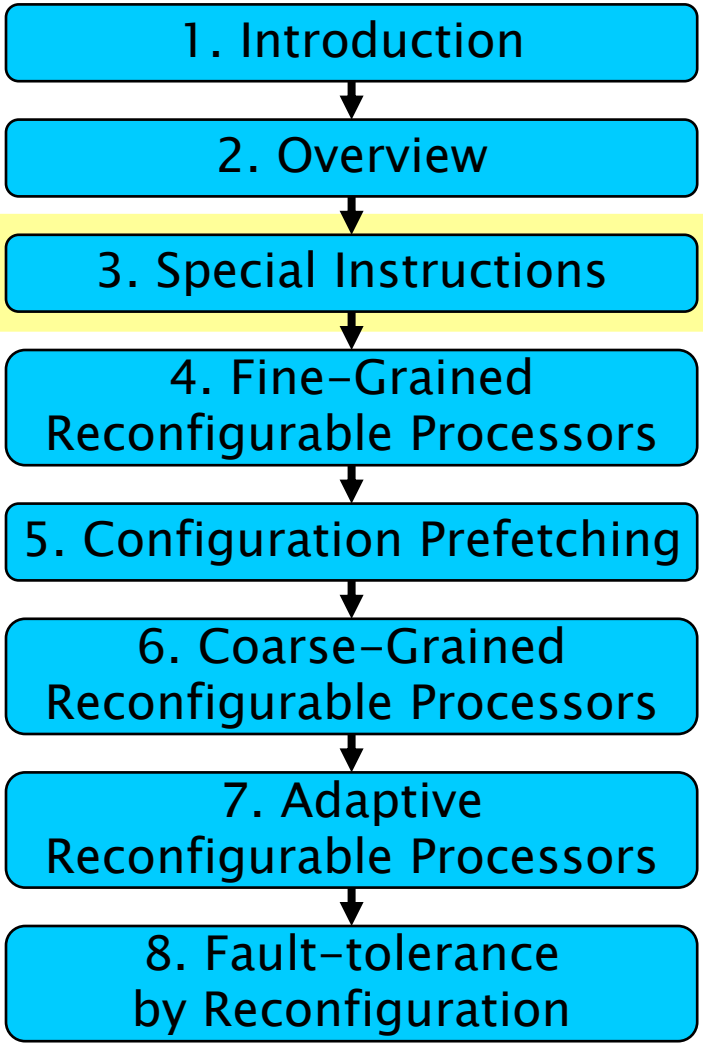
Karlsruhe Institute of Technology
Institut für Technische Informatik
Chair for Embedded Systems – Prof. Dr. J. Henkel

Reconfigurable and Adaptive Systems (RAS)

3. Special Instructions

or: How to use the reconfigurable fabric

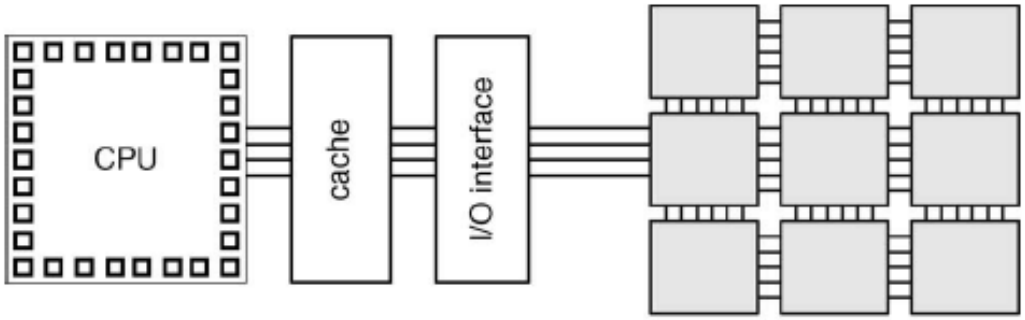
RAS Topic Overview



- Connecting the reconfigurable fabric
- Special Instructions
  - Input Data
  - Control
  - Coding
  - Operand Passing
  - Automatic Detection
  - Configuration Thrashing

3.1 Connecting the reconfigurable fabric

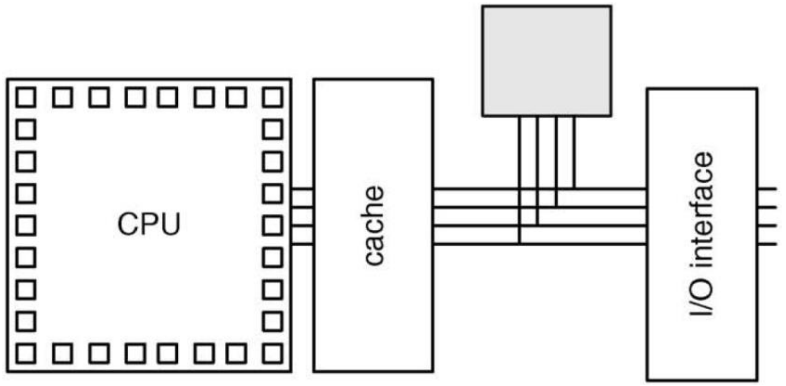
- Different alternatives exist to connect the reconfigurable fabric with the (core-) CPU:
- External stand-alone processing unit
  - Off-chip reconfigurable fabric, connected using I/O pins
  - So-called 'loosely coupled'
  + Can be used to connect the reconfigurable fabric with general purpose processors on existing ICs
  + Fabric & CPU may execute in parallel (like GPU in PCIe card)
  – Very high communication overhead
  – No access to CPU-internal information (e.g. registers)
    - All data has to be transferred via the data bus



src: [TCW+05]

Attached processing unit

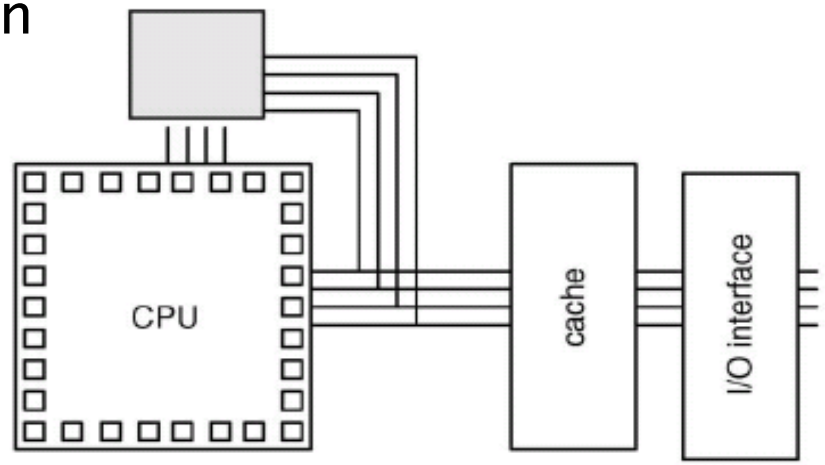
+ Faster on-chip communication
+ Can be used to connect the reconfigurable fabric with general purpose processors
+ May access external shared memory when using a Cache coherency protocol
  - Often, the control signals for such a protocol are not provided to I/O pins; thus the off-chip coupling (previous approach) typically cannot use shared memory
- Still relatively high communication overhead and no access to CPU-internal information
- Requires developing a new IC



src: [TCW+05]

Coprocessor

- Similar to the attached processing unit
- \+ Additionally using dedicated Coprocessor interface
  - Providing dedicated control signals to start/interact with the calculations
  - Might provide an interrupt that informs about completion of operation (no need for polling the coprocessor)
- – Same drawbacks as attached processing unit



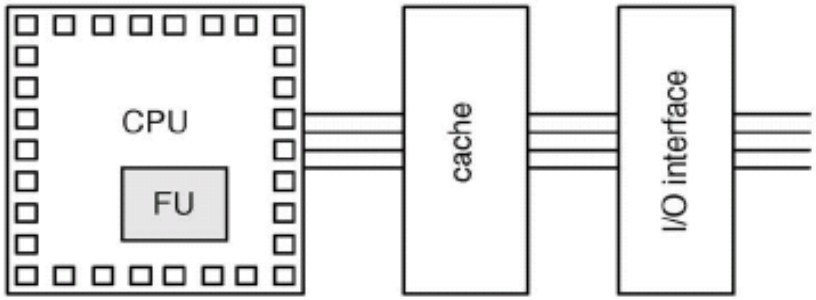
src: [TCW+05]

Reconfigurable Functional Unit (RFU)

- So-called tightly coupled
- Using an embedded reconfigurable fabric
- CPU = 'core processor' with RFU

+ Very low communication overhead (accessed like an ALU or any other FU)
+ High data bandwidth due to access to the CPU internal information (e.g. the register file) in addition to the memory access

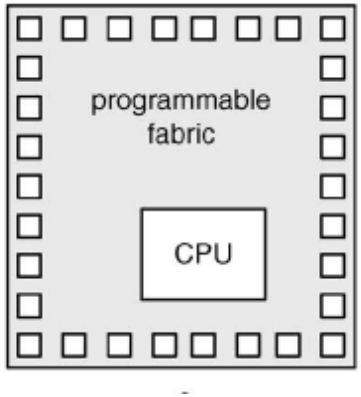
– Requires developing a new IC
– Requires modifying the CPU architecture



src: [TCW+05]

Processor embedded in a reconfigurable fabric

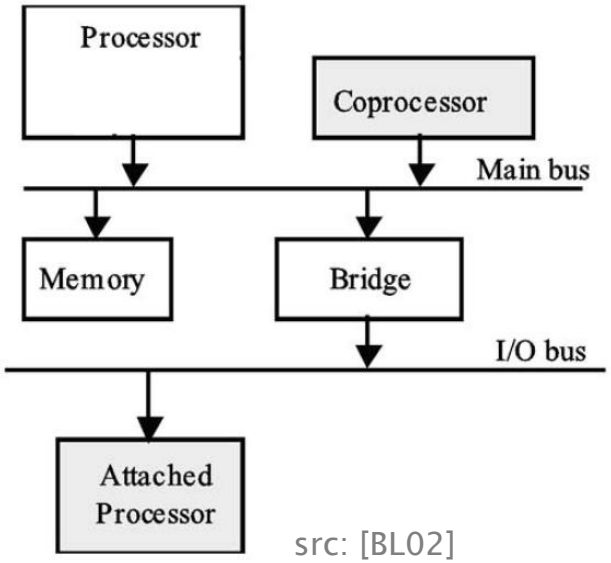
- Processor may be soft core (i.e. synthesized / implemented for the fabric) or a hard core (i.e. an ASIC element within the fabric)
- \+ High availability (e.g. using standard FPGAs), i.e. no IC needs to be developed
  - Often used to simulate the Coprocessor or RFU approach, i.e. can be used for loose- and tight coupling
- – May have noticeably reduced frequency of the CPU
- – May require modifying the CPU architecture



src: [TCW+05]

Summarizing: Loosely coupled reconfigurable fabric

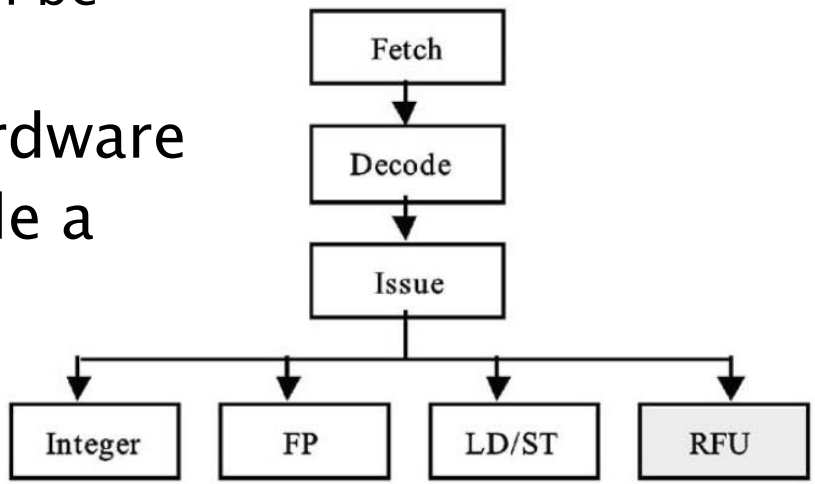
- The communication overhead of the loosely coupled architectures (external/internal attached processor and coprocessor) limits their applicability
  - E.g. 50 cycles communication cost for the round trip in PRISM-I
- The speed improvement using the reconfigurable logic has to compensate for the overhead of transferring the data
  - This can be the case in applications where a huge amount of data has to be processed using a simple algorithm that fits in the reconf. fabric
- Their main benefit is the ease of constructing such a system using a standard processor and standard reconfigurable fabric
- Another benefit of this approach is that the microprocessor and reconf. fabric can work on different tasks at the same time



src: [BL02]

Summarizing: Tightly coupled reconfigurable fabric

- Communication costs are insignificant
  - As a result, it is easier to obtain an increased performance for a wider range of applications
- Design costs for this approach are higher
  - It is not possible to use standard components
- Multiple RFUs can be connected to the core pipeline
  - i.e. the reconfigurable fabric can be partitioned into multiple RFUs
- Amount of reconfigurable hardware is limited to what can fit inside a chip
  - Limits the performance gain



src: [BL02]

3.2 Special Instructions (SIs)

- The Instruction Set Architecture (ISA) is an abstraction level between the hardware and the application
- Each processor provides a so-called core ISA, i.e. the ISA that is implemented with the regular FUs
- ASIPs and Reconfigurable Processors extend this core ISA by additional instructions, so-called Special Instructions (SIs)
  - Also called Custom Instructions or Instruction Set Extensions
- For the application programmer it appears as an assembly instruction
- In Reconfigurable Processors an SI is implemented using reconfigurable hardware
  - Using fine-grained or coarse-grained reconfigurable fabrics
  - Typical for tight coupling; can also be used for loose coupling

Instruction Set Architecture and Micro Architecture

- Instruction Set Architecture (ISA)
  - Type: RISC, CISC, VLIW, …
  - Bit widths of data and address busses
  - Number and size of visible registers (there might be further registers, e.g. pipeline registers, or register windows)
  - Instruction formats, actual instructions, addressing modes etc.
  - A range of (virtual) memory addresses; stack handling
  - Interrupt and exception handling
  - Different privilege levels (e.g. for OS support)
  - Function Calls (recommendations/rules for callers and callees)
- The ISA serves as the interface to the compiler
- Microarchitecture
  - (Reconfigurable) Functional units
  - Memory hierarchy; Cache architecture
  - Branch prediction
  - Bus Systems; Periphery

3.3 SI Input Data

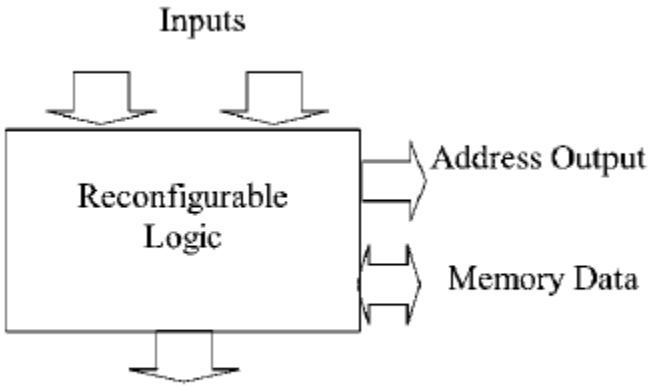
- Stream-based Special Instructions:
  - They process large amounts of data in sequence (like a continuous video sequence)
  - Only certain tasks can benefit from this type
  - Most of them are suitable for a coprocessor approach
  - Examples: finite impulse response (FIR) filter, packet processing (e.g. checksum, encryption) etc.
- Chunk-based Special Instructions:
  - Not streaming huge amount of data but working on larger parts of data (more than can be provided via the registers)
  - E.g. DCT on a 16x16 Macroblock of a video frame

SI Input Data (cont'd)

- Element-based Special Instructions:
  - Read small amount of data at a time (usually from internal registers) and produce small amount of output
  - Can be used in almost all applications (they impose fewer restrictions on the applications' characte-ristics)
  - The obtained speedup is usually smaller
  - Example: bit reversal, multiply accumulate (MAC), variable length coding (VLC), trigonometric functions (sin, cos), e^x,…

SIs with Memory Access

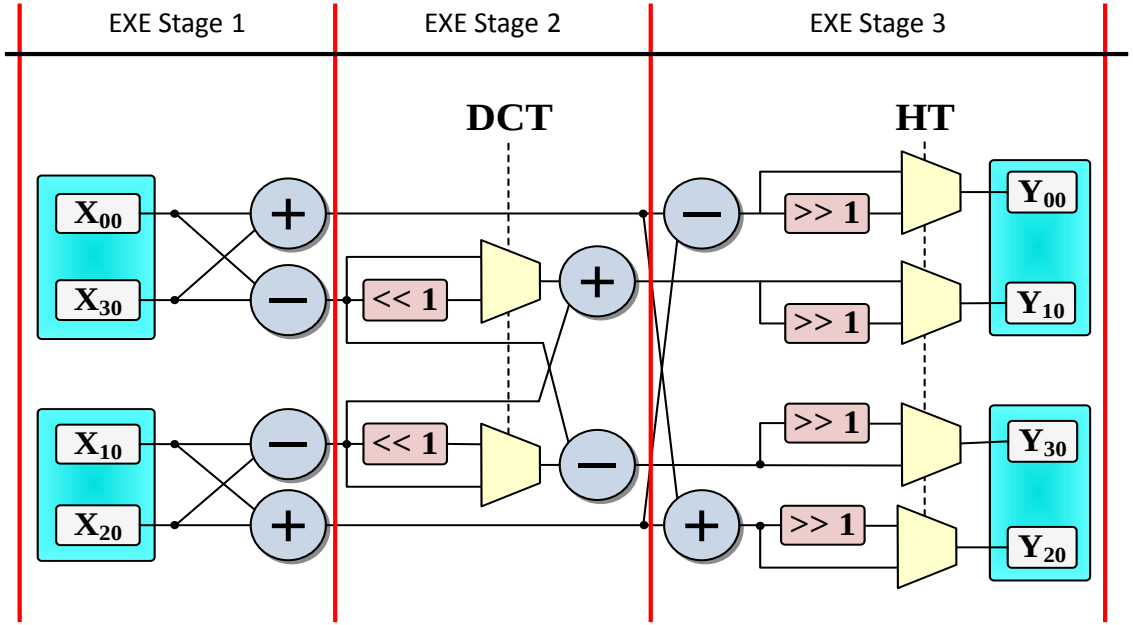
- Complex addressing schemes are used in many multimedia applications
  - SIs would make these accesses more efficient
- Providing access to memory hierarchy allows implementing specialized load/store operations or stream-based operations
  - The SI as an address generator: The SI logic is used to generate the next address; address is fed to the standard load/store unit
  - The SI uses the data memory: data is accessed and processed by the SI
- If the SI can access memory, it is important to maintain consistency between the SI accesses and the processor accesses



src: [BL02]

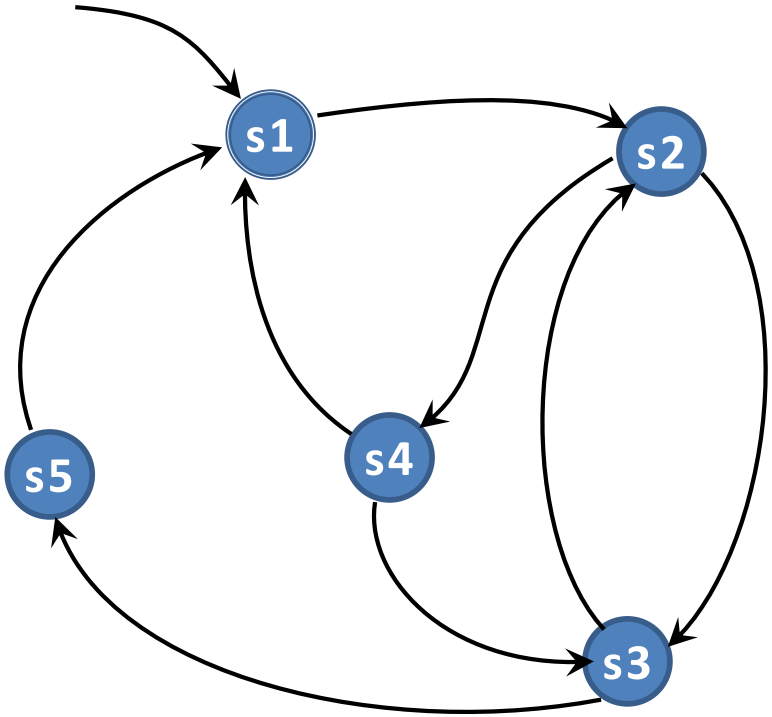
3.4 SI Control

- SIs often perform complex operations that cannot be completed in a single cycle
- Either use a pipelined implementation (multiple SIs can reside in different stages of the RFU at the same time
- Or use a multi cycle implementation
- A pipelined implementation provides higher throughput, but is more compli-cated in case a shared resource is accessed (e.g. main memory)



SIs with Internal state

- State machine can control the execution sequence of a particular SI execution
- Can also be used to pass information from one SI execution to another
  - E.g. 'context adaptive' coding, or a partial sum
- Allows sharing a common resource (e.g. hardware block or memory access) among multiple states



Fixed/variable execution length (i.e. latency of one SI execution in cycles)

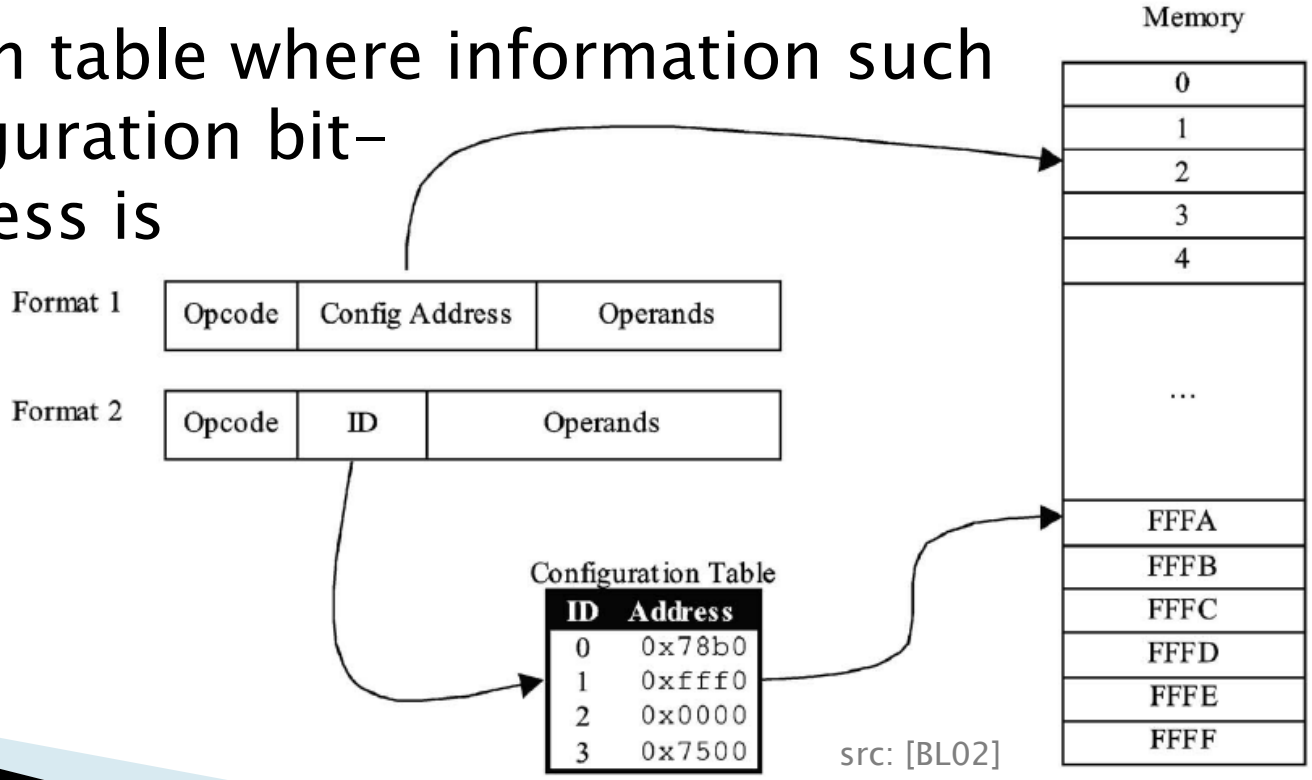
- 'Variable' is problematic for a VLIW processor
  - E.g. due to memory access or calculation that depends on the input data
  - Unknown duration would result in pipeline stalls with a potentially large performance loss
- For a super-scalar processor, variable execution length can be dealt efficiently
  - The RFU can be used similar to one of the standard FUs by reservations stations
  - Multiple RFUs can be dealt by multiple reservation stations

3.5 Instruction Coding

- Generally, SIs for reconfigurable processors are created at compile time
- SIs are embedded as assembly instructions to the application
  → need unique opcode when assembling
- Number of free opcodes is typically limited due to 32-bit instruction word length
- For SIs, the opcode is typically partitioned into two parts:
  - Format Identifier: A value in the regular opcode fields (i.e. those that are also used by the core ISA) determining that this is an SI (not declaring which one)
  - SI Identifier: determines which SI is meant

Meanings of the SI Identifier

- Address: The memory address of the configuration bitstream for the instruction; example: DISC, MOLEN etc.
- Instruction Number: An identifier that indexes a configuration table where information such as the configuration bit-stream address is stored; example: OneChip98, RISPP etc.



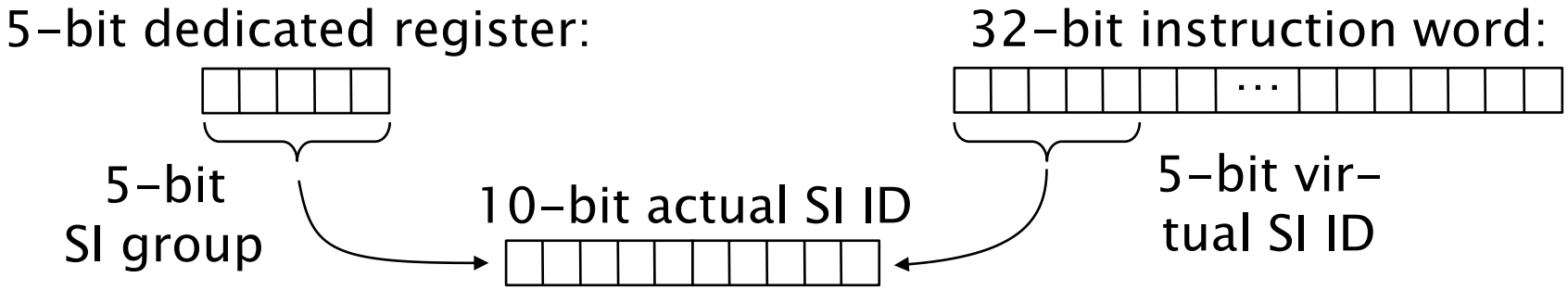
src: [BL02]

Meanings of the SI Identifier (cont'd)

- Using an 'Address' identifier needs significantly more bits, but the number of SIs is not limited by a table
  - Drawback: less bits are available to provide information about operands
- For the 'Instruction Number' identifier, the number of supported SIs can be increased if the contents of the table can be changed at run time
  - Drawback: complex task for the compiler, i.e. which SIs shall be available in the table at which time? This demands a control-flow analysis

Virtual SI Identifiers

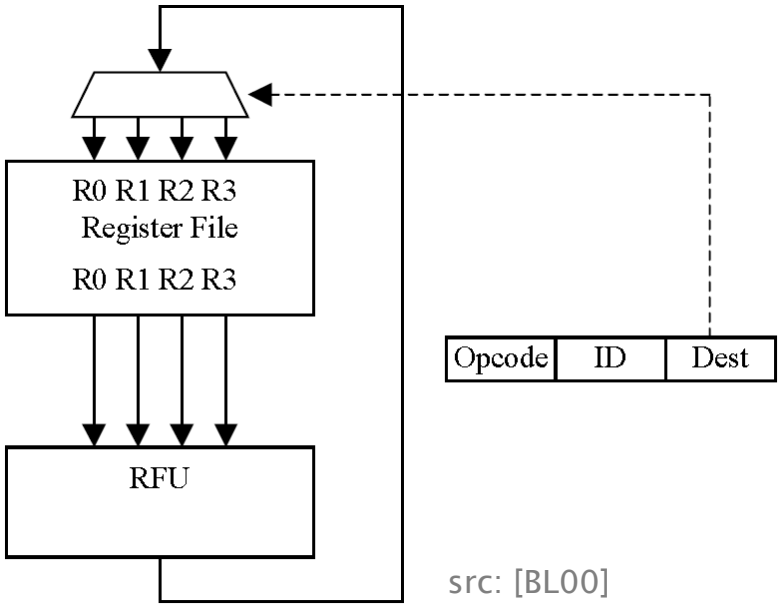
- Approach for extending the number of supported SIs (or reducing the number of opcode bits): Virtual SI Identifiers
  - Provide a dedicated register, accessible from the application
  - SI Identifier corresponds to the concatenation of the bits in the register and the bits in the application binary



  - Use so-called Helper Instruction to read/write the dedicated register
  - The resulting 'actual SI ID' can be used as bitstream address or as instruction number (i.e. table pointer)

3.6 Operand passing

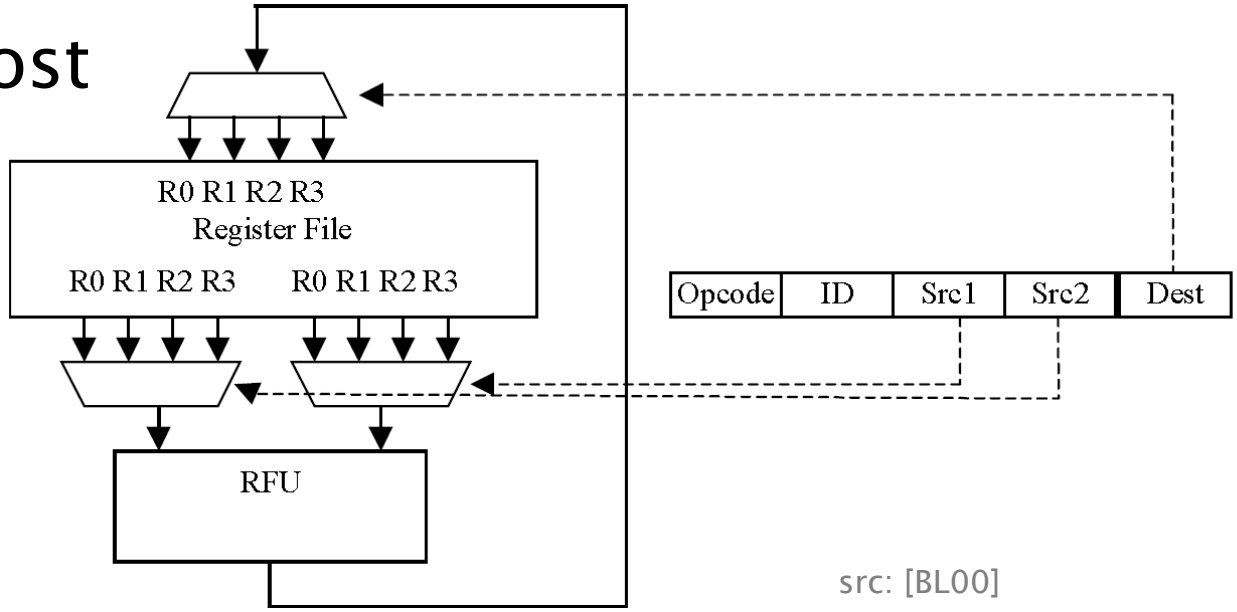
- The instruction word also specifies the operands to be passed to the SI
  - Can be immediate values, registers, etc.
  - Can determine the source and/or destination of the operation
- Hardwired Operand Coding:
  - The contents of all registers (or a fixed subset) are sent to the SI
  - The registers actually used depend on the particular SI
  - This allows the SI to access more registers but makes the register allocation more difficult for the compiler
  - Example: Chimaera (all eight registers from the register file can be accessed simultaneously)



src: [BL00]

Fixed Operand Coding

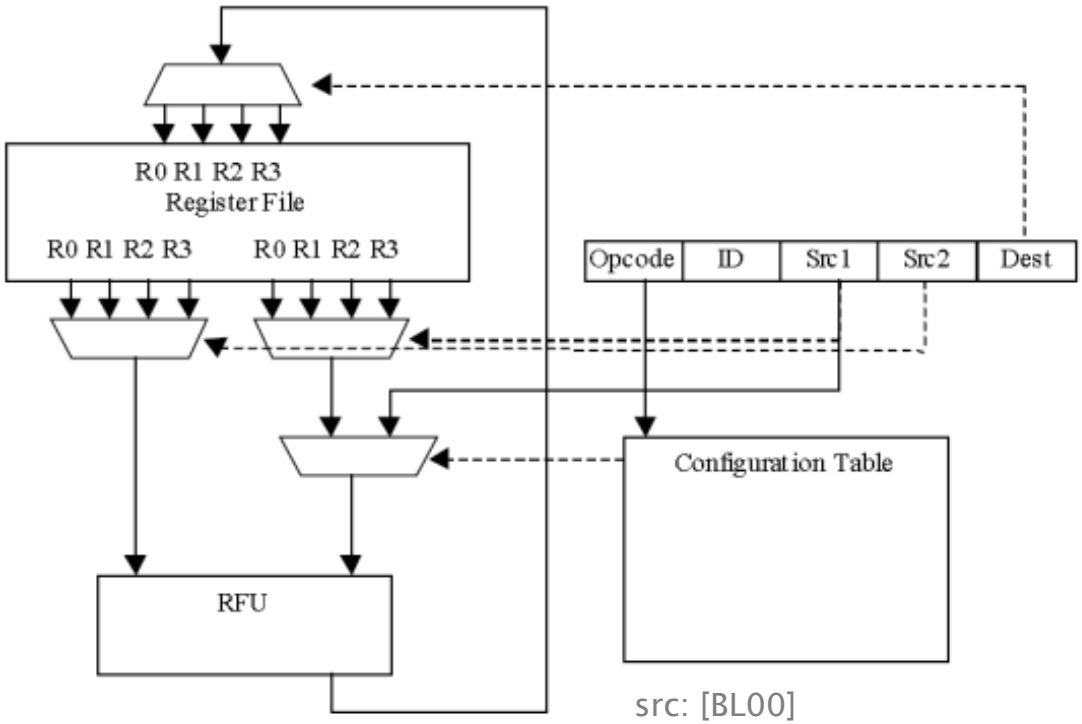
- The operands are in fixed positions in the instruction word and are of fixed types
- Different encoding formats would have different opcodes
- This the most common case
- Example: OneChip98



src: [BL00]

Flexible Operand Coding

- The position of the operands is configurable
- The degree of configuration can be very broad
- A configuration table can be used to specify the decoding of the operands
  - E.g. register addr. or immediate value
- Example: DISC, RISPP



src: [BL00]

Register File Access

- SIs may use a dedicated register file or a shared register file (i.e. shared with other instructions / functional units)
- A dedicated register file needs less ports than if it was shared (some data might come from the general-purpose register file, some from dedicated register file)
  - Data has to be explicitly transferred to dedicated register file
  - Natural solution for Coprocessor coupling
  - Example: MOLEN
  - Reduces HW complexity but complicates code generation due to heterogeneity of registers
- Currently, most reconfigurable processors use a shared register file
  - This might change when more superscalar and VLIWs are used as core processor

Case Study: RISPP

- Based upon Sparc-V8 ISA
  - Using Virtual SI identifiers (altogether 1024 SI IDs)
- Using shared register file with 4 read ports & 2 write ports
  - The 2 write ports are implemented as multi-cycle write back
  - Still beneficial in comparison to 1 write port, e.g. consider a multi-cycle operation with 2 results (e.g. div_mod) that would have to be called twice otherwise (i.e. div, mod)
- Using flexible operand coding
  - Different Formats declare particular parts of the instruction word as register address or immediate etc.
- Providing SI memory access
  - Variable execution length
  - SIs not pipelined (could conflict with memory access)
  - Allows internal state to control the SI flow and to reuse resources
- Typically chunk-based or element-based SIs

Case Study: RISPP (cont'd)

- Overview: Sparc-V8 ISA:

Format 1 (op=1): Call

	op	disp30
bit#	31	30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Format 2 (op=0): SETHI & Branches

	op	rd	op2	imm22
	op	a cond	op2	disp22
bit#	31	30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0		

Format 3 (op=2 or 3): Remaining Instructions

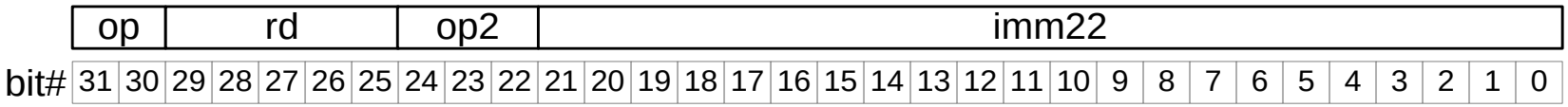
	op	rd	op3	rs1	i=0	asi	rs2
	op	rd	op3	rs1	i=1	simm13	
	op	rd	op3	rs1	opf		rs2
bit#	31	30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14			13	12 11 10 9 8 7 6 5	4 3 2 1 0

src: "The SPARC architecture manual, version 8"

Case Study: RISPP (cont'd)

- Usage of 'Format 2':

Format 2 (op=0):



Value for *op2*	SPARC V8 Allocation	RISPP Extension
0	UNIMP	Helper instructions*
1	unused	SI without register write back
2	branch on integer cond.	
3	unused	SI with one register write back
4	set register high 22 bits; NOP	
5	unused	SI with two register write back
6	branch on floating-point cond.	
7	branch on coprocessor cond.	

*: To support the concept, e.g. used to switch the virtual SI identifier

Case Study: RISPP (cont'd)

- Extension of 'Format 2':

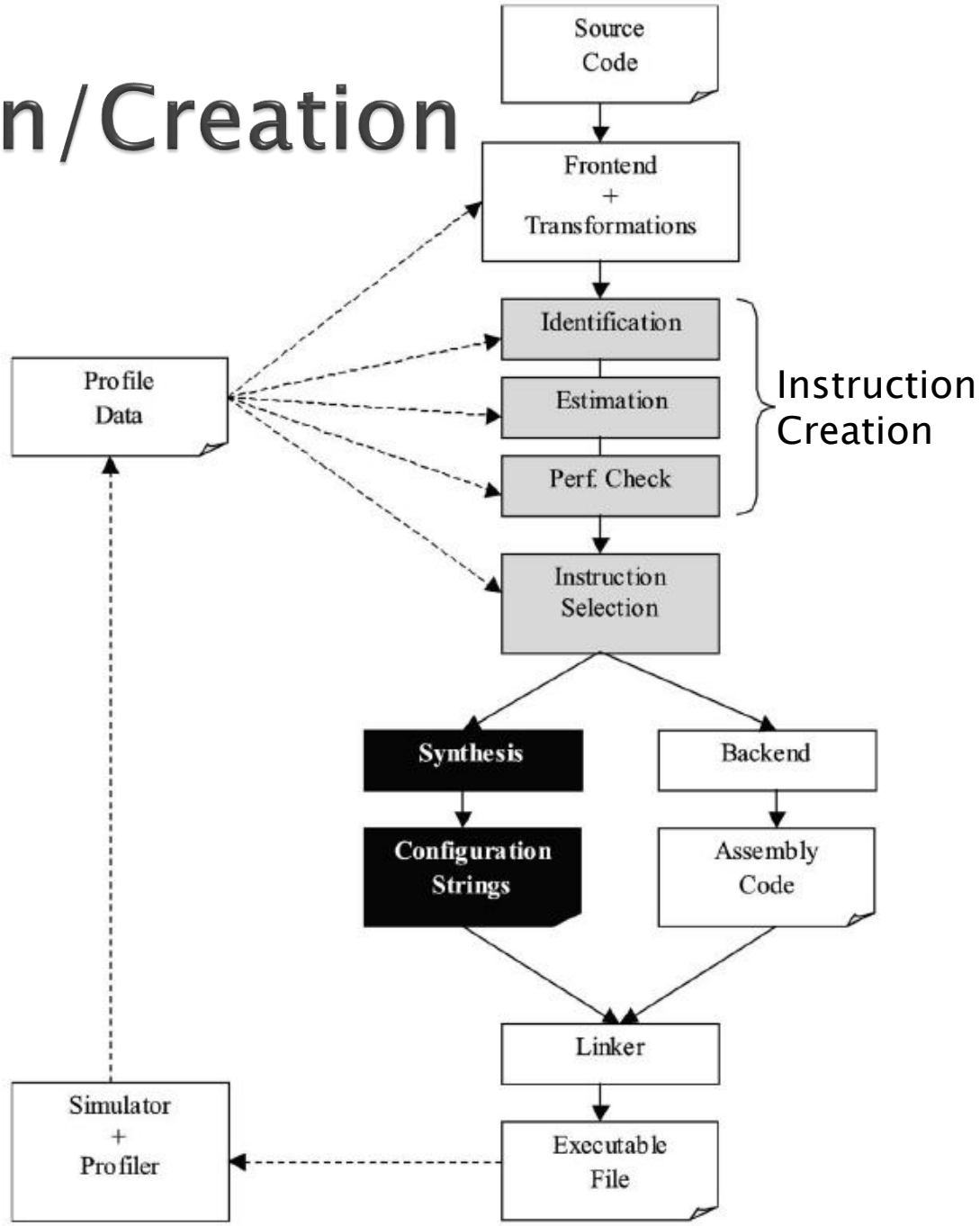
	op										**imm**																		
	0	0	rd					**op2**			0	0	**si_op**					rs5					rs4					rs2		
	0	0	rd					**op2**			0	1	**si_op**					rs5					rs4					imm5		
	0	0	rd					**op2**			1	0	**si_op**					rs5					imm5					imm5		
	0	0	rd					**op2**			1	1	**si_op**					rs5					imm10							
bit#	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

op2 determines the register write back	***imm*** determines the input immediate
001 : no write back	00 : no immediates
011 : rd write back	01 : rs2 used as 5-bit immediate
101 : rd & rs5 write back	10 : rs2 and rs4 used as two 5-bit immediates
	11 : rs2 and rs4 used as 10-bit immediate

src: [BSH08]

3.7 SI Detection/Creation

- White: traditional compiler blocks
- Grey: New Techniques for SI Creation
  - Pruning is used to reduce the amount of candidates
- Black: traditional HW Synthesis



src: [BL02]

Code Filtering

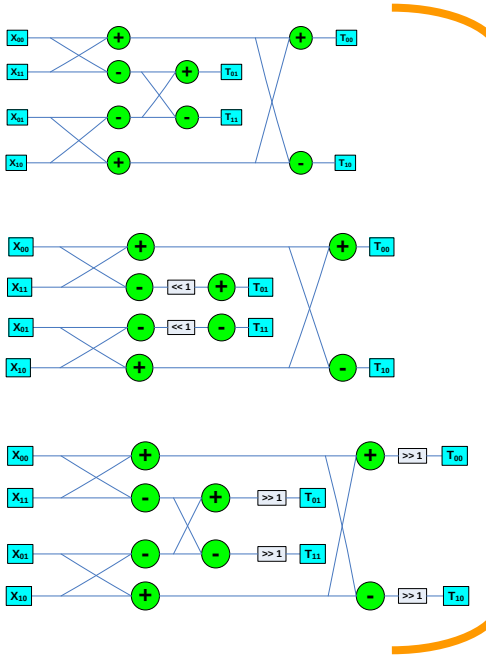
- Objective: Reduce the amount of code to be processed
- Manual identification:
  - Programmer annotates the code with special directives (e.g. ‘pragma’)
  - Used to identify the places that shall be optimize
- Static identification:
  - Compiler analyses the code to determine candidates for potential optimization (e.g. loops)
  - Potentially limited, since the execution profile of some programs depends on their input data
- Dynamic identification:
  - Code is initially compiled without Special Instruction identification
  - Optimization potential is identified by executing code on real data (profiling)
  - Most time consuming approach, but can achieve the best results
  - Important: relies on significant and relevant data set to get good estimates

Instruction Creation

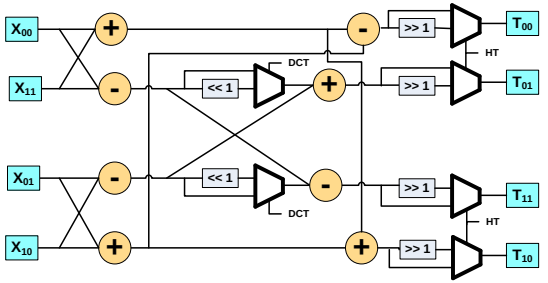
- Identification:
  - Based on analyzing the control/dataflow graph
  - Create new instructions by grouping operators or by performing code transformations
  - Result: a description of the new instructions
- Parameter estimation (instruction characterization):
  - The instruction description is processed and important parameters, like instruction latency, size, reconfiguration time etc. are estimated
- Instruction performance check:
  - Checks whether or not the new instructions improve the execution time of the code block

Instruction Merging

- Reduces the number of different instructions that have to be placed in the RFU → reduces the reconfiguration time
- Can result in increased latency, size, or other parameters



- Consider constraints
  - Max. size of data path
  - Number of I/O signals
  - Number of control signals
- Increase reusability
  - Combine similar data paths (MUX)



Instruction Selection (Configuration Scheduling)

- For ASIPs: select one globally optimal instruction set
- Here: select multiple locally optimal instruction sets and schedule the reconfigurations of the RFU along all control paths
  - Local for individual hot spots
- Since the compiler cannot optimize all control paths, it has to estimate the most common path and optimize it (using profiling), considering that
  - reconfiguring the RFU takes time and resources and
  - the performance of the code depends on what instructions are configured into the RFU
- In some compilers, no selection is done
  - Instead, for each block, the instructions that optimize the block are implicitly selected
  - This can lead to solutions in which the processor spends most of its time reconfiguring the RFU (so-called 'Thrashing', described a few slides later)

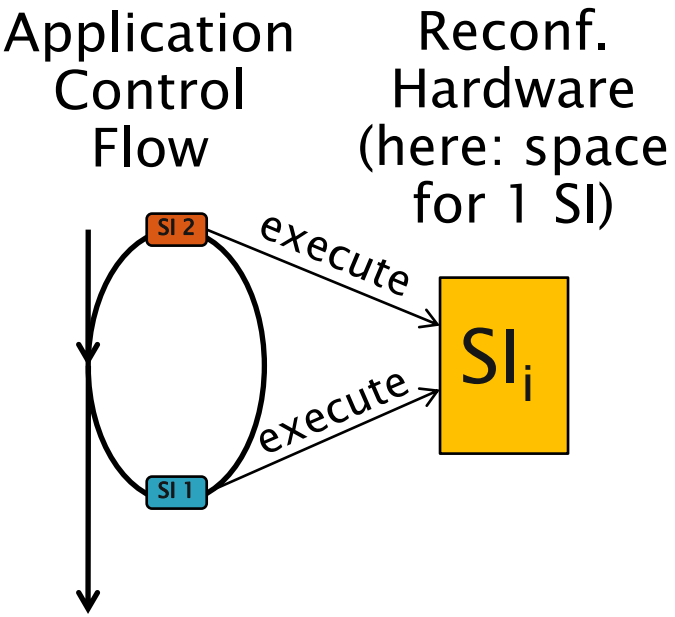
Retargetable Backend

- The intermediate representation is marked with information concerning where to use SIs
- The backend has to schedule the instruction and assign a set of operand registers
  - Alternative: the SIs have to be explicitly used by the programmer (inline assembly) if the compiler is not able to use them automatically
- If the reconfigurable logic runs asynchronously to the core processor, the backend needs to insert synchronization instructions

3.8 SI Configuration Thrashing

- Typical problematic scenario:
  - Within one inner loop more Special Instructions (SI) are demanded than fit to the reconfigurable hardware at the same time, i.e. #SIs > FPGA capacity
  - Per loop iteration, some SIs need to be replaced to load the next SI → frequent reconfiguration, i.e. configuration thrashing
  - → Performance is significantly slower than without SIs (depends on reconfiguration time and SI execution time)

Example:



→ two reconfigurations per loop iteration, i.e. configuration thrashing

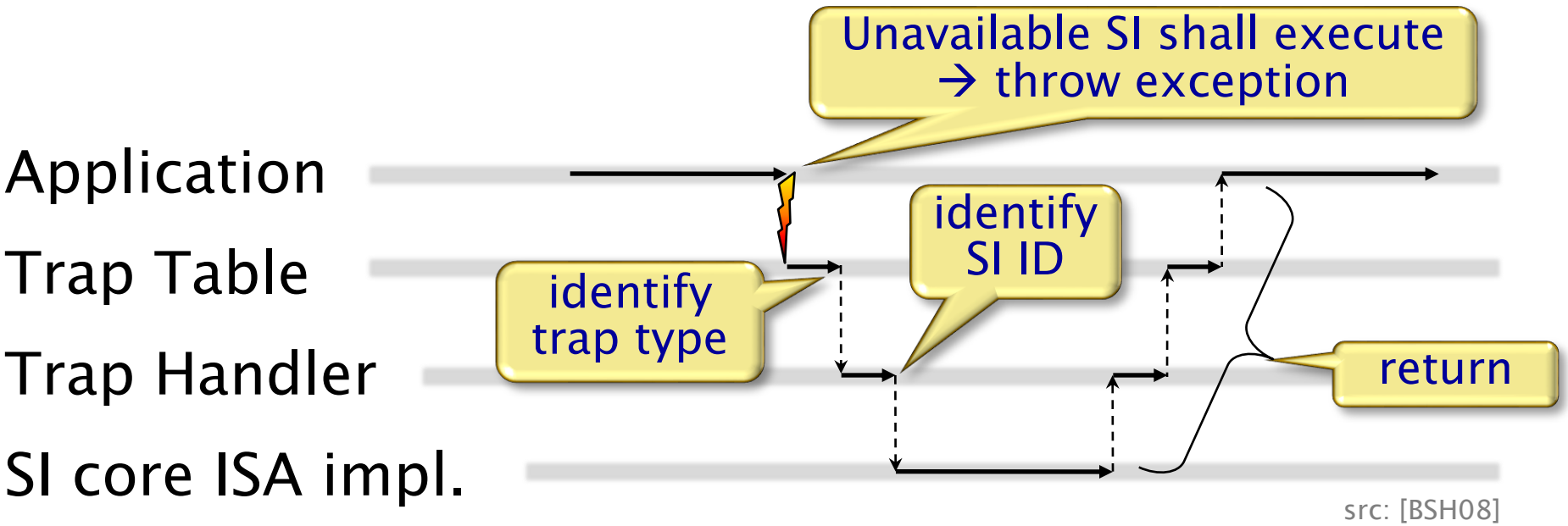
SI Configuration Thrashing (cont'd)

- Simple Solution:
  - Assumption: at compile time the capacity of the FPGA is known (i.e. how many SIs fit to the FPGA at the same time)
  - Then: predetermine, which SI 'candidates' (of a hotspots) shall be realized as SIs (implemented on the FPGA) and which shall not (implemented with the core ISA)
    - Assure that all SIs of a hotspot fit to the FPGA at the same time
- Drawback:
  - Upgrading to larger FPGA is inefficient (application won't use it)
  - Downgrading to a smaller FPGA might introduce thrashing again
  - What if the reconfigurable logic has to be shared among multiple competing applications?
    - Then it is not known how many SIs of one task fit to the FPGA at the same time
  - Note: these are similar problems like in VLIW architectures (code needs to be recompiled when number of Slots changes)

Core ISA Implementation

- Better approach: For each SI provide an alternative implementation using the core ISA
- If the hardware Implementation for the SI is not available when the SI is demanded then trigger the core ISA implementation
  - Using a trap, e.g. 'unimplemented instruction exception'
  - Typically used for CPUs that may or may not have floating point support etc.
- Let a run-time system decide which SIs shall be implemented with the reconfigurable hardware and which shall use the core ISA

Trap Handler implementation



src: [BSH08]

- Overhead in RISPP implementation: 38 cycles per trap (incl. everything), compared to 544 cycles for e.g. SI core ISA implementation (for SATD, i.e. Sum of Absolute Transformed Differences)
- Further benefit: core ISA implementation may bridge the reconfiguration time (i.e. avoid stalling)

Conditional Branch

- Alternative Solution: Conditional Branch
  - Introduce new Helper Instruction that tests the availability of the SI implementation
  - Example:

    IF *(hardware implementation for SI_x available)*
    THEN
        use SI assembly instruction
    ELSE
        use core ISA to implement SI functionality
    END IF
- Drawback: Introduces Overhead independent of whether or not the SI implementation is available
  - The trap handler only introduces overhead when the SI implementation is not available (then it is slow anyway)

Comparison of 'Conditional Branch' and 'Trap Handler'

- Important Parameters
  - How often is the SI executed?
    - If it is executed rather seldom (in comparison to other SIs), then maybe its hardware may never be reconfigured (or reconfigured rather late) and thus most SI executions will be implemented using the core ISA → Conditional Branch advantageous
  - SI execution time?
    - If the SI execution time is rather short (e.g. 30 cycles using core ISA) then an overhead of 38 cycles for the trap handler would dominate the execution time → Conditional Branch advantageous
- For SIs that are executed very often and that have a long core ISA execution time, the Trap Handler approach is advantageous

Case Study: RISPP

- Problem: The trap handler needs to identify which SI was executed and which parameters were passed
- Example: Identifying the SI ID
  - Read the SI instruction word
    - Read 'return register' of trap (pointing to the instruction after the SI), calculate address of SI from that and load the 32-bit SI instruction word
  - Extract the 5-bit SI ID
    - Load a mask (an immediate value) into a register, 'and' it to the 32-bit of the SI and shift the result to the LSB
    - Load the 5 bit from the dedicated register for the virtual opcode
    - Combine both values (logical 'shift' and 'or' operation)
  - Similar for the parameters (registers, immediate values etc.)
  - Altogether: very large overhead
- Solution: additional Helper Instructions to accelerate this process
  - The micro architecture knows the SI ID after the SI execution, it only needs to be provided to the trap handler via another Helper Instruction

Case Study: RISPP (cont'd)

- Red highlights show the new Helper Instr.
- Loads all possible register / immediate combinations
  - Could be optimized towards specific SIs
  - Exploits the availability of 2 write ports in register file, i.e. "regmov1" stores 2 of the (at most) 4 input registers

```
void unimp_handler() {
   int si_id, regsav, g1, psr, rd1, rd2;
   int rs1, rs2, rs4, rs5, imm10, imm5_1, imm5_2;
   asm( "mov %g1, g1"                  // save %g1 register
        "mov %psr, psr"                // save CPU status
        "siid si_id"                   // load SI identifier
        "regmov1 rs1, rs2"             // load input registers
        "regmov2 rs4, rs5"
        "imov5 imm5_1, imm5_2"         // load immediates
        "imov10 imm10"
   );
   switch (si_id) {                    // jump to cISA execution
   case 0x2A:                          // one showcase SI opcode
        ...                            // here comes cISA execution
        break;
        default:
        regsav = 0;                    // set amount of write backs
   break;
   }
   asm( "mov psr, %psr"                // restore CPU status
        "mov g1, %g1"                  // restore %g1 register
        "nop"
        "regsav rd1, rd2, regsave"     // SI register Write Back
        "restore"                      // restore register window
        "jmpl %l2, %g0"                // set jump target
        "rett %l2 + 0x4"               // and return from handler
   );
}
```

src: [BSH08]

References and Sources

[TCW+05] T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk and P.Y.K. Cheung: “Reconfigurable computing: architectures and design methods”, IEE Proc.-Comput. Digit. Tech., vol. 152, no. 2, pp. 193-207, March 2005.

[BL02] F. Barat, R. Lauwereins: “Reconfigurable Instruction Set Processors from a Hardware/Software Perspecitve”, IEEE Transactions on Software Engineering, vol. 28, no. 9, pp. 847-862, September 2002.

[BL00] F. Barat, R. Lauwereins: “Reconfigurable Instruction Set Processors: A Survey”, IEEE International Workshop on Rapid System Prototyping (RSP), pp. 168-173, June 2000.

[BSH08] L. Bauer, M. Shafique, J. Henkel: “A Computation- and Communication Infrastructure for Modular Special Instructions in a Dynamically Reconfigurable Processor”, IEEE 18th International Conference on Field Programmable Logic and Applications (FPL), pp. 203-208, September 2008.