Flexible Instruction Processors

Wayne Luk & Peter Y.K. Cheung
Imperial College

CASES 2000

“Not So General” Purpose Processors

- GPPs lose performance when dealing with non-standard operations and data
- Particularly apparent in embedded systems
  - Cell phones
  - Medical appliances
  - Digital cameras
  - PDAs

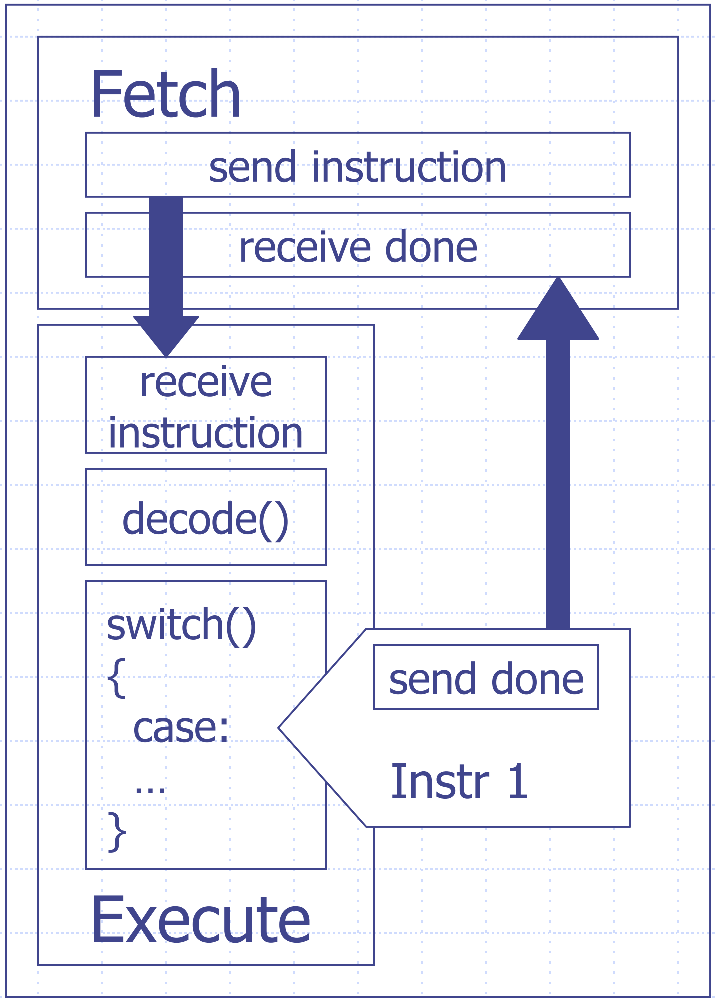
Two roads to customization

- Augment GPPs with programmable logic
  - Couple standard processor (ARM, MIPS) with an FPGA fabric
  - Fixed processor instruction set
  - FPGA implements custom instructions
- Implement them in FPGAs
  - Customize instructions at compile time or at run time

Flexible Instruction Processors

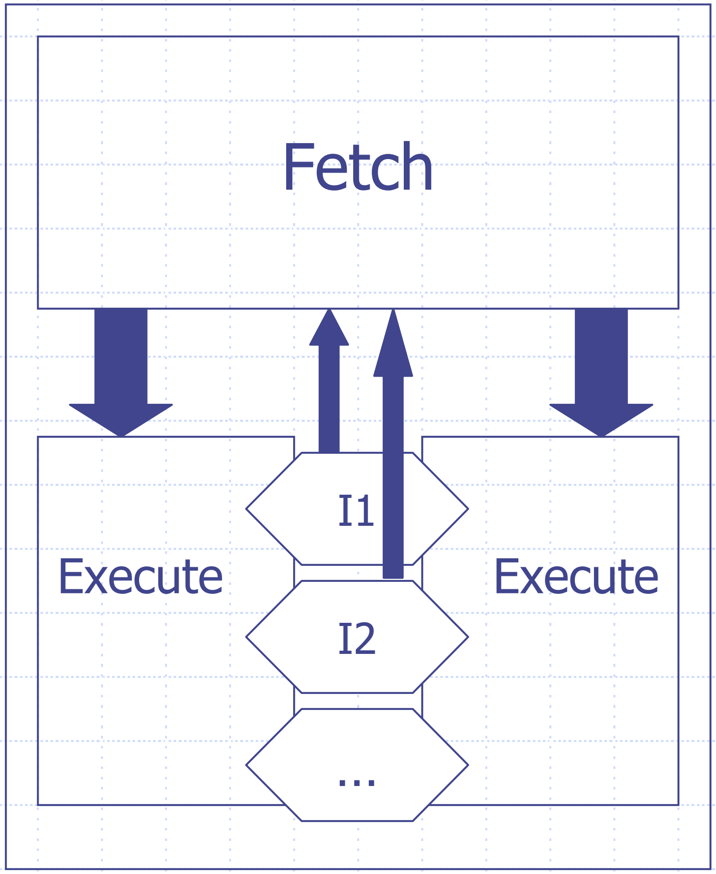
- FIPs are assembled by parameterized processor templates
- Allows customized hardware to be exposed as new instructions
- Critical resources can be adjusted according to domain requirements
- Can implement same instruction set with different design tradeoffs

FIPs Template



- Templates are modules connected together by communicating channels
- Can instantiate different styles of processors
  - Stack, register, etc.
- Customizations
  - Removal of resources
  - Changing data widths
  - Vary degree of pipelining
- Implemented in Handel-C

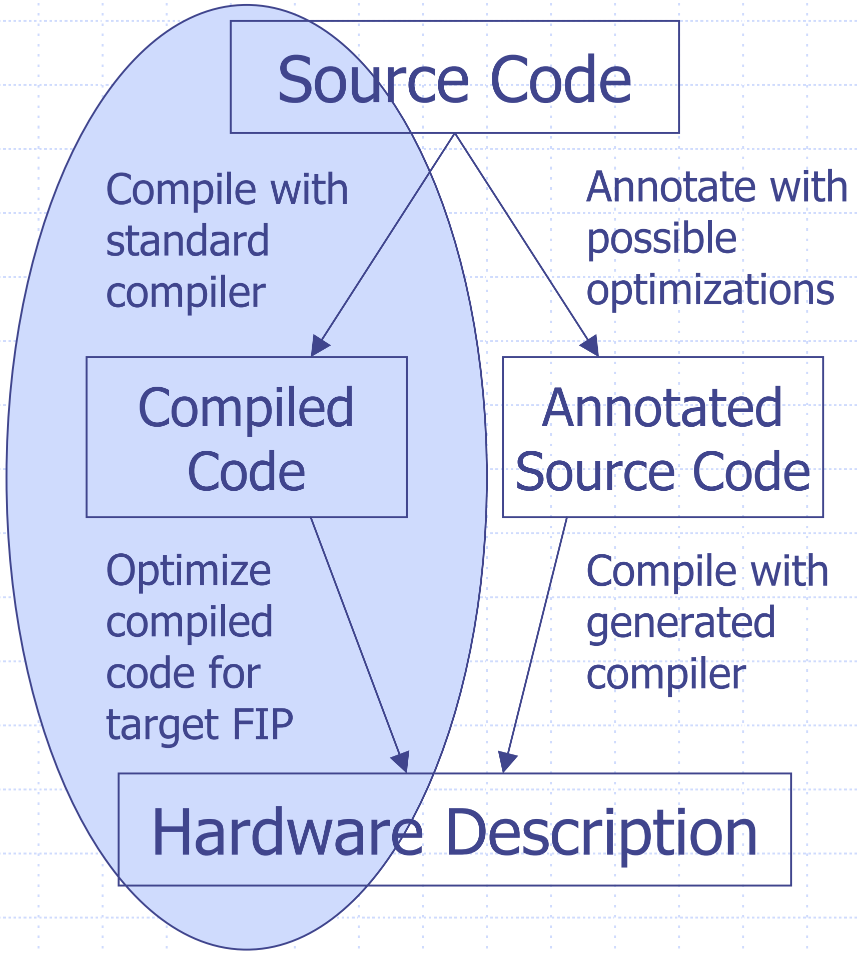
Enhancing Processor Templates



- Can group modules differently to achieve different processor styles
- Superscalar
  - More than one execution module
- Hybrid
  - Different execution modules (stack/register)

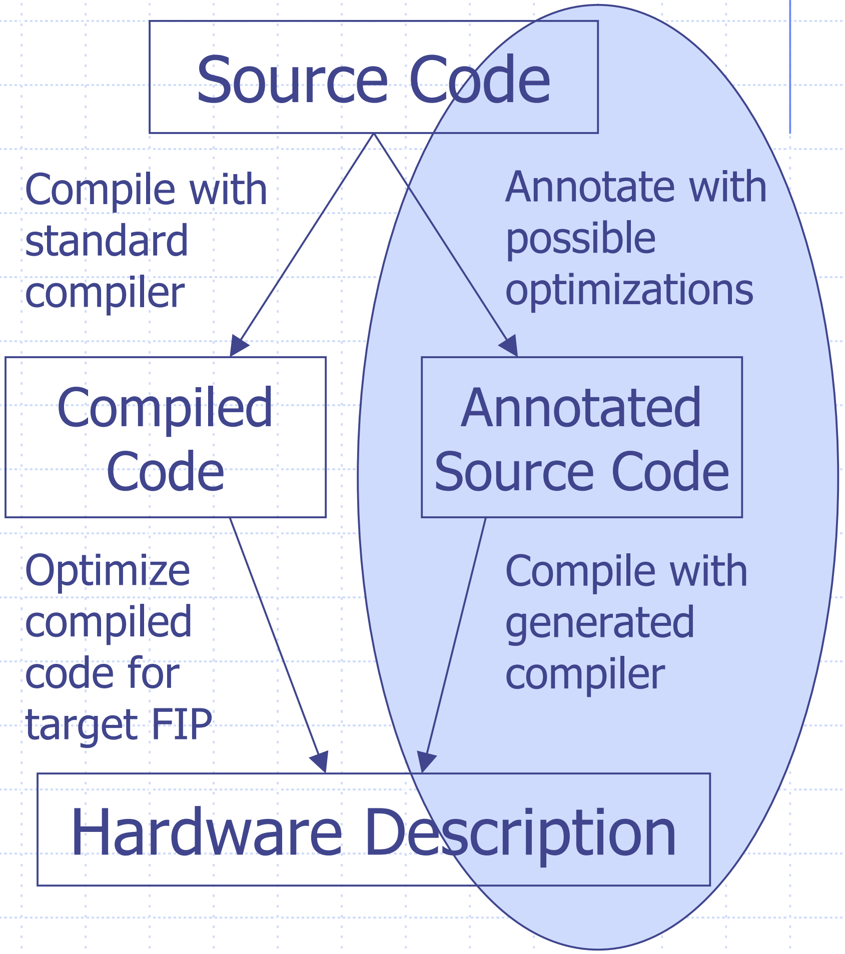
Compilation Strategy

- Optimizes known target code to known target FIP
- Reorganizes code to take advantage of ILP
- Similar to Java's Just-In-Time compilation


Source Code
Compile with standard compiler
Annotate with possible optimizations
Compiled Code
Annotated Source Code
Optimize compiled code for target FIP
Compile with generated compiler
Hardware Description

Compilation Strategy

- Annotate source code with profile information
  - Frequency of instruction use
  - Common instruction groups
  - Shared resources
- Use annotations to drive optimization of code and FIP
- Compile with a FIP-specific compiler



Source Code
Compile with standard compiler
Annotate with possible optimizations
Compiled Code
Annotated Source Code
Optimize compiled code for target FIP
Compile with generated compiler
Hardware Description

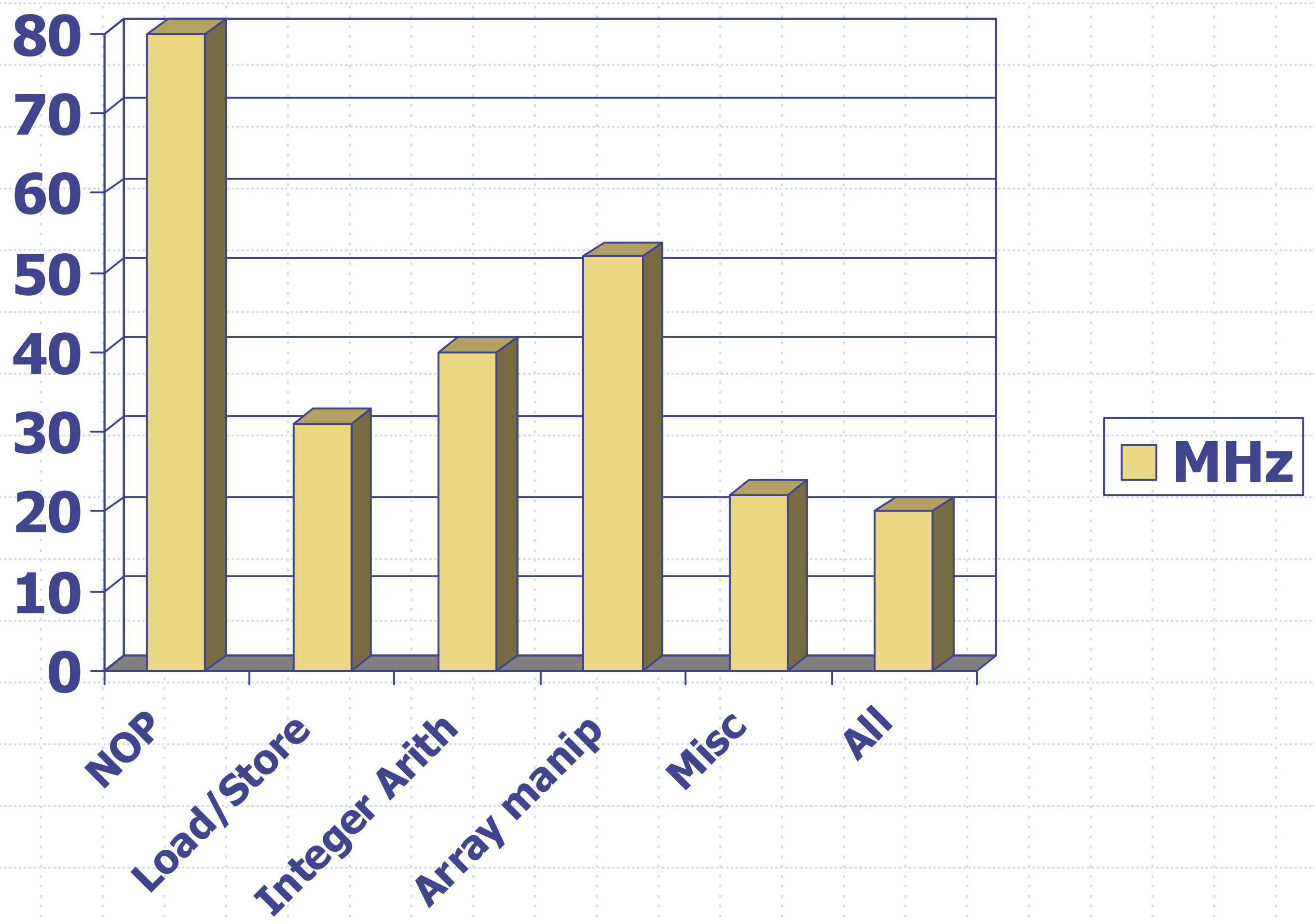
FIP Optimizations

- Custom instructions
  - Direct hardware implementations can be exposed by custom instructions
  - Reduces number of fetch-decode instructions
  - Increases area
- Processor-specific
  - Can replicate much-used resources
  - JVM: allow concurrent reads of top of stack
  - JVM: allow reads of top ‘n’ elements

Implementation: JVM

- Implemented FIP-based Java Virtual Machine
  - #1: segregated resources that are shared
    - Good area utilization and the expense of routing congestion
  - #2: two-stage pipeline & only shares stack and main memory
  - #3: deeper pipelines for certain instructions and ‘register’ style improvements
    - Top-of-stack registers are replicated – writes are done back to stack directly, replicated registers updated during the fetch cycle
    - Most instructions have 4 stages, some 5-6 stages
    - Routing is pipelined

Results


80
70
60
50
40
30
20
10
0
NOP
Load/Store
Integer Arith
Array manip
Misc
All
MHz

Results

Devices	CaffineMark	Speedup
Software JVM	502	1
FIP JVM	1019	2x
Pipelined FIP JVM	3665	7x
GMJ30501SB	13332	27x

FIP implemented in Xilinx Virtex @33MHz
GMJ30501SB == 200MHz ASIC

Impact of custom instructions

- Link list insertion (100 links)
  - Direct hardware: 2.3ms
  - FIP-JVM: 6.4ms, 22 instructions
  - ASIC: 1ms
- Adding ‘insert link’ instruction
  - 12 cycles to complete
  - Saves 10 cycles and 10 fetch/decodes

Implementation: AES

Device	Cycles/Block	Slices/BRAM	Mbps/MHz
Software	340	N/A	0.4
FIP (I)	99	1770/2	1.3
FIP (II)	32	1393/10	4
Hardware I	11	460/10	11.5
Hardware II	1	2679/82	129.6

FIP (I) direct implementation of transforms
FIP (II) table-lookups for transforms
Hardware I Spartan II 100-6
Hardware II Virtex-E 812-8

Optimizations

- Opcode chaining
  - Store intermediate results in temporary registers
  - Removes push/pop and alleviates sequential nature of stack-based machines
- Galois Multiplication is single instruction
  - No procedure call overhead
  - 8.5 cycles instead of 30
- Introduce predicate registers to allow instructions to execute in parallel
  - 6 cycles instead of 30