Mochamad Asri
Tokyo Institute of Technology

---

**What do we do?**
- Propose Efficient Processor Fetch Mechanism

**How do we do?**
- Employing Large Entry of Instruction Register
  - Instruction Register is a register file that stores the most frequently executed instructions
  - Performing Unique Binary Translation

**How did it turn out?**
- Code size could be diminished up to 54%

---

**Research Background**
- The Advancement of Transistor Technology
  - Processors are embedded in many electronics systems
- Strict design constraints in Embedded Processor
  - Power Consumption
  - Performance
  - Area Size
  - Code Size
- In order to address each design issue:
  - Identification of inefficiencies in some parts of mechanism is needed

---

**Re-Thinking Instruction Fetch Mechanism**
- Fetch Logic Optimization is one of important design issues
  - In Fact, 36% of Strong ARM Power is consumed by Fetch Logic
  - The Most Energy-Consuming Part!
- Moreover, conventional fetch mechanism is inefficient at least in two aspects:
  - Use the same length
  - Most instructions use only a fraction of available length
  - Access from the same storage (IC hit or ROM access)
  - Only small subset of instructions account for the majority of reference
- How do come up against these inefficiencies?

---

**Instruction Register**
- Instruction register\(^2\) is proposed to reduce those inefficiencies.

**So, What is Instruction Register?**
- A Register File that stores the most frequently executed instructions.

<table>
<thead>
<tr>
<th>Instruction Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>


**Note that although addi R3, R3, 4 and addi R3, R3, 8 apply the same addi instruction, they will not be regarded as same instruction since they have different binary code.**

**Architectural Position**
- Instruction Register in the Decode stage.

**The Proposed Idea**
- Based on the previous work:
  - The larger entry number of IR
  - The bigger percentage of IR-resident instructions

**With the help of** binary translation:
- Possible to reduce Code Size!

**Target:**
To realize efficient fetch mechanism that leads to further reduction of code size!
Preliminary Experiment

- Investigate percentage of IR-resident instructions when the number of IR entry are expanded
  - To see if it is high or not
- Evaluate by using SimMips MIPS Processor Simulator
- SPECINT2006 (bzip2, gcc, mcf, omnetpp) as the benchmark

---


Trade-off Consideration

- On Average, IR-Resident Instructions covers:
  - About 83% of total instruction for 256-entry IR
  - About 90% of total instruction for 512-entry IR
  - About 96% of total instruction for 1024-entry IR
  - About 97% of total instruction for 2048-entry IR
- Decided to employ 1024-entry of IR
  - It covers almost the same high percentage of instructions with 2048-entry
  - While at the same time it costs only half hardware of 2048-entry

Instruction Register + Binary Translation

- It is found that large IR stores most of program instructions
- Considering this fact:
  - If we apply 1024-entry IR
    - Only need 10 bits for reference.
    - But, how to reduce the code size?

Let's make the most of binary translation!
Binary Translation: IR-Resident

- IR-resident case:
  - Translate to 11 bits
  - Express 1024-entry IR with 10 bits
  - Use the remaining 1 bit as a flag bit,
  - Decide whether an instruction is IR-resident or not

IR-resident Instruction Code

<table>
<thead>
<tr>
<th>Flag</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>000101000</td>
</tr>
<tr>
<td>0</td>
<td>000001101</td>
</tr>
<tr>
<td>1</td>
<td>000001100</td>
</tr>
<tr>
<td>1</td>
<td>000001011</td>
</tr>
<tr>
<td>1</td>
<td>000001000</td>
</tr>
<tr>
<td>1</td>
<td>000001001</td>
</tr>
</tbody>
</table>

Flag Code

<table>
<thead>
<tr>
<th>10-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

IR-resident Instruction Code

<table>
<thead>
<tr>
<th>Flag</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>010100000</td>
</tr>
<tr>
<td>1</td>
<td>010100001</td>
</tr>
<tr>
<td>0</td>
<td>010100000</td>
</tr>
<tr>
<td>1</td>
<td>010100000</td>
</tr>
</tbody>
</table>

Flag Code

<table>
<thead>
<tr>
<th>32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Binary Translation

**The Novel Fetch Mechanism**

- IF (Instruction Fetch) Stage
  - If (flag) refer to IR
  - 10 bits
  - 1024 Entries

- First Half of ID (Instruction Decode) Stage
  - Translated Binary

- If flag is 0, then concatenate the current 10-bit and 2 following 11-bit codes so that it forms 32-bit instruction
- Else, refer to Instruction Register

Simulation Environment

- Processor Simulator
  - Evaluate by using SimMips
- Benchmark
  - SPECINT2006 (bzip2, gcc, mcf, omnetpp)
  - Apply train data set
**Experimental Result**

- Code size reduced up to maximum 48%
- In average, code is diminished up to 54%
  - Previous work only reduced up to 81%

![Graph showing code reduction rate for different applications]

**Conclusion**

- Proposed Efficient Fetch Mechanism
  - Employing 1024-entry of IR
  - Performing novel binary translation
- Succeeded in reducing code size up to 54%
  - The previous work reduce only up to 81%

**Future Works**

- Further Evaluation
  - Power Consumption
  - Area Consumption
  - Performance Evaluation
- FPGA Verification
  - Implement Processor in FPGA

Thank you!