BEAM: Bus Encoding Based on Instruction-Set-Aware Memories

Yazdan Aghaghiri and Massoud Pedram
University of Southern California
Farzan Fallah
Fujitsu Laboratories of America

Outline

- Introduction
- Previous Techniques
- Instruction-Set-Aware Memories
- Instruction Address Bus
- Data Address Bus
- Conclusion
Memory Bus

- Wide and highly capacitive
- High switching activity
- Significant power consumption

Bus Power Minimization

- Physical Layer
  - Materials
  - Signaling and voltage levels

- Data Link Layer
  - **Encoding** → Reduce bus activity → Reduce power
    - Redundant lines may be employed during bus encoding
Transition Cost

Transition cost of an instruction or data address is defined as:
- the number of bit transitions that occur on the bus from the current address to the next address

Transition cost can be calculated by bit-wise Exclusive-OR operation:
- Transition Cost = Current_Address ⊕ Next_Address
  For Example: Current_Address: 1000, Next_Address: 1101, Transition Cost=3

Instruction and data addresses can be sent on separate buses or multiplexed on the same bus

Embedded Processors

- I/O power is significant
  - Bus encoding can be quite effective
- Low system clock rate (less than 200 MHz)
  - Delay of the extra encoding/decoding hardware is more tolerable
- Many of the embedded processors don’t have internal caches or they bypass it for some memory accesses
  - An example of the latter case is streaming applications in which each access fetches external memory
Guidelines

- A large portion of the switching activity on an address bus may be eliminated by making use of the sequential access behavior of instruction
  - T0, T0-Concise or Offset-Xor-SM are quite effective in achieving this goal
- We have to be careful about the hardware overhead
  - So as not to offset the power reduction due to reduced activity on the bus by the power increase due to the encoder/decoder logic
- We must determine the dominant source of remaining activity on the bus

Why SimpleScalar

- Based on MIPS-IV Instruction Set Architecture
- Simulator, Compiler and SPEC2000 pre-compiled binaries are available
- Instruction, Data and Multiplexed address traces for system w/o cache
- No out-of-order execution
**SPEC 2000 Benchmarks**

**Transition Cost for Different Categories of Instructions**

**Percentage of Control Flow Instructions**

**Offset-XOR-SM**

This block extracts the XOR distance of addresses; at the same time it eliminates the transition cost for sequential addresses.

LSB-Inv(x) : if x is negative, we invert all bits except the MSB. Small offsets always cause small number of activities.

<table>
<thead>
<tr>
<th>Offset</th>
<th>LSB-Inv(Offset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0101, (+5)</td>
<td>0000 0101</td>
</tr>
<tr>
<td>FFFF FFFF, (+1)</td>
<td>8000 0000</td>
</tr>
<tr>
<td>FFFF FFFF, (-5)</td>
<td>8000 0004</td>
</tr>
<tr>
<td>8000 0000</td>
<td>FFFF FFFF</td>
</tr>
</tbody>
</table>
Transition Costs after Offset-XOR-SM

- Costs associated with sequential instructions and not taken branches are completely eliminated

Transitions Costs after Applying Offset-XOR-SM

ISA-Aware Memories

- Memory has the instruction and its address
- Add simple hardware to predict/calculate the target of control flow and memory access instructions inside the memory itself
- Processor supervises this process:
  - When the memory exactly calculates the address, the processor will not intervene
  - When the memory only predicts the address, the processor will have to validate or possibly correct this prediction based on its own calculation
  - When memory is unable to calculate/predict the address, the processor will send the target address on the bus
Control Flow Instructions

- **BRANCH**
  - Offsets are known. Memory still needs to predict the outcome of the branch, i.e. taken or not taken

- **JUMP and JUMP AND LINK**
  - Offsets are known. Target address is deterministically calculated

- **JUMP REGISTER and JUMP AND LINK REGISTER**
  - Target is determined by the value of a register
  - Offsets cannot be determined at compile time

Branch Prediction

- Memory **Predicts** the target of the branch locally
- Processor both calculates and predicts the target
  - Sends no signal to memory if prediction matches the calculation and hence it is OK
  - Sends an error signal if the prediction is incorrect
- For each miss prediction, one transition occurs on the bus
- Memory seeks validation of its target prediction by monitoring the bus

- **No Signal:** If exact calculation or correct prediction on the memory side
- **Error Signal:** If incorrect prediction, e.g. send a limited-weight-code like 1000...0 by transition signaling
Memory Decoder

- The previous instruction, the previous address, and some additional information received from the processor are used to determine the next address

\[ \text{Current Address} = \text{ISA AWARE UNIT} + 1 \]

Results of Predicting BRANCH, J and JAL

<table>
<thead>
<tr>
<th>Level of the optimization</th>
<th>% Reduction in switching activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Encode sequential instructions using Offset-XOR-SM</td>
<td>80.0%</td>
</tr>
<tr>
<td>Above scheme plus</td>
<td></td>
</tr>
<tr>
<td>- Predict (in memory) the targets of branches assuming that they are taken</td>
<td>86.2%</td>
</tr>
<tr>
<td>- There is one transition on the bus for each miss prediction, i.e., each not taken branch</td>
<td></td>
</tr>
<tr>
<td>Above scheme plus</td>
<td></td>
</tr>
<tr>
<td>- Calculate target addresses of J and JAL</td>
<td>92.1%</td>
</tr>
</tbody>
</table>

- After these optimization, JR instructions become the major source of activity on the bus
What to Do for JR and JALR

- JR is used for
  - Function returns
  - Case statements

- JALR
  - Used for pointer-based function calls
  - Very rare
  - JR penalty can be decreased if we store the return address of the corresponding function. For example

0x:40040  JAL Function_1
0x:40048  Instruction_2

JAL jumps to the beginning of function_1 and links the next address. The return address is stored in memory.

Later on when Function_1 returns to execute instruction_2:
Function_1: ...
JR Link_register
Use the stored address to predict the value of the Link_register in memory.

Predicting Return Addresses (JR)

- Use a stack in the processor and in the memory to store the return addresses for each JAL

- For each JR:
  - In the processor, if it is a return instruction, freeze the bus. Otherwise, send the target address
  - In the memory, if there is no activity on the bus, pop the return address from the stack. Otherwise, use the received address as the target

A stack implemented both in the memory and the processor:

<table>
<thead>
<tr>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>404008</td>
</tr>
<tr>
<td>46510a</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>JR Link_register</td>
</tr>
</tbody>
</table>

In memory:

<table>
<thead>
<tr>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>JR Link_Register</td>
</tr>
</tbody>
</table>

No activity on Bus: pop the address and use it as target

Address received on Bus: Use the received address and don’t touch the stack
**Stack Size**

- A circular stack performs better than a linear stack
  - When the depth of nested function calls is very large, linear stack will become totally useless because of overflow
- 10 to 15 entries would be enough if a circular stack is used

![Graph showing number of transitions caused by JR for different sizes of circular stack](image)

**Results**

<table>
<thead>
<tr>
<th>Level of the optimization</th>
<th>% Reduction in switching activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encode sequential instructions using Offset-XOR-SM</td>
<td>97.3%</td>
</tr>
<tr>
<td>Predict/Calculate BRANCH, J and JAL as previously stated plus</td>
<td></td>
</tr>
<tr>
<td>Predict JR using a 15-entry circular stack</td>
<td></td>
</tr>
</tbody>
</table>

- The remaining activity is due to the miss prediction of branch instructions
- Each incorrect prediction costs one transition on the bus
- Better prediction schemes can be used to eliminate more of the unwanted transitions
  - About 1% improvement using an 8-bit global branch predictor
  - Using more advanced techniques up to 99.5% reduction in activity can be achieved. This is however not practical because of huge hardware costs
Predicting Data Addresses

- Calculate the memory address that an instruction is going to access based on the Rs field of the instruction.
  
  \[ \text{MEM-INS } Rt, (Rs)\text{OFF} \]
  
  For example for load instruction: \[ \text{MEM}(Rs+\text{OFF}) \to Rt \]

<table>
<thead>
<tr>
<th>MEM-INS</th>
<th>Rt Field</th>
<th>Rs Field</th>
<th>OFF</th>
</tr>
</thead>
</table>

- The OFF field is extracted from the instruction.
- Memory should have its own Rs register.
- For correct prediction, the two copies of the Rs register should be synchronized, however:
  - The value of the Rs doesn’t change very often.
  - The processor sends the target address when memory cannot calculate it and, at that time, the memory updates its Rs.

Memory Decoder

- Address is either taken from the bus or is calculated locally.
- Received address from the bus, if any, will be used to update the shadow register file.
Shadow Register File

- The decoder and the encoder have a special shadow register file.
- Shadow registers are updated with each memory access instruction (for the $R_s$ that has been employed in that instruction).
- If $R_s$ does not change between two consecutive memory accesses, no activity will occur on the bus.
- An example for $R_8$ is shown below.

<table>
<thead>
<tr>
<th>$R_8$ in processor</th>
<th>$R_8$ in memory</th>
<th>Instruction</th>
<th>Bus Status</th>
<th>Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>100</td>
<td>Mem Access using $R_8$</td>
<td>No Activity</td>
<td>OK</td>
</tr>
<tr>
<td>100→80</td>
<td>100</td>
<td>Modifies $R_8$</td>
<td>No bus access</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>100→80</td>
<td>Mem Access using $R_8$</td>
<td>Activity</td>
<td>FAIL</td>
</tr>
<tr>
<td>80</td>
<td>80</td>
<td>Mem Access using $R_8$</td>
<td>No Activity</td>
<td>OK</td>
</tr>
</tbody>
</table>

Reducing Shadow Registers

- Most of the time, only a small number of registers are used by the compiler as pointers to access memory.
- We can therefore reduce the number of registers in memory to 4 instead of 32.
- Register values are cached in this 4 entry register file by direct mapping.
- Activity reduction drops from 82.4% to 75.4%, but hardware cost on the memory side is significantly lowered.

```
Mem_Ins R12, R10(12) Prediction Fail, Cache R10 in entry #3
... Prediction OK
Mem_Ins R14, R10(-16) Prediction Fail, R10 is evicted and R6 is stored instead
Mem_Ins R14, R6(-16) Prediction Fail, R10 is evicted and R6 is stored instead
... R6 is invalidated
ADD R6, R13, R7
```
Memory Design

- Memory should be aware of the Instruction Set format
- Memory should be designed for a family of similar architectures
- Most embedded processors use a RISC architecture with rather similar ISA
- Programmable registers may be used to store some info about the Instruction Set format during initialization phase, so the additional hardware in the ISA-aware memory can be programmed (customized) at initialization time

Hardware Analysis and Power Evaluation

- Hardware analysis:
  - Was performed by assuming separate instruction and data address buses
  - Was done for memory only (1.5v, 0.18u technology)
    - For the instruction address bus: J, JAL and BRANCH instructions are predicted
    - For the data address bus, a four-entry shadow register file was used

<table>
<thead>
<tr>
<th></th>
<th>Instruction Codec</th>
<th>Data Codec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num of Literals</td>
<td>686</td>
<td>720</td>
</tr>
<tr>
<td>Area ( * 1000 λ² )</td>
<td>343</td>
<td>588</td>
</tr>
<tr>
<td>Num of Gates</td>
<td>311</td>
<td>528</td>
</tr>
<tr>
<td>Original Bus Power (uW)</td>
<td>5205</td>
<td>20050</td>
</tr>
<tr>
<td>Bus Power with BEAM (uW)</td>
<td>416</td>
<td>6055</td>
</tr>
<tr>
<td>Power of BEAM memory Codec (uW)</td>
<td>384</td>
<td>1144</td>
</tr>
<tr>
<td>Codec power + Bus Power with BEAM</td>
<td>780</td>
<td>7199</td>
</tr>
<tr>
<td>Power Saved with BEAM (uW)</td>
<td>4421</td>
<td>12851</td>
</tr>
<tr>
<td>Percentage Saving over Bus</td>
<td>85%</td>
<td>54%</td>
</tr>
</tbody>
</table>
Conclusions

- We proposed an effective and low overhead technique for instruction address and data address bus encoding.
- Some blocks in the processor can be eliminated when using this encoding, but memory must be made ISA-aware.
- This technique resulted in 97% reduction for instruction addresses and 82% reduction for data addresses.
- Next step is to extend the basic approach to processors with internal caches.