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

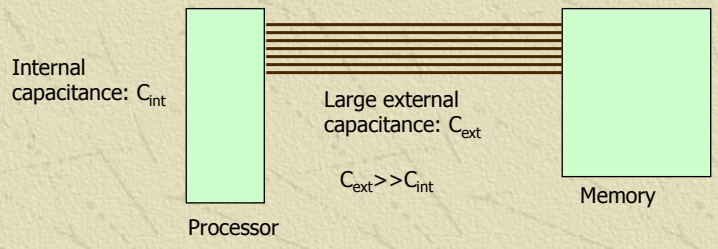
Yazdan Aghaghiri and Massoud Pedram
University of Southern California
Farzan Fallah
Fujitsu Laboratories of America

Outline

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

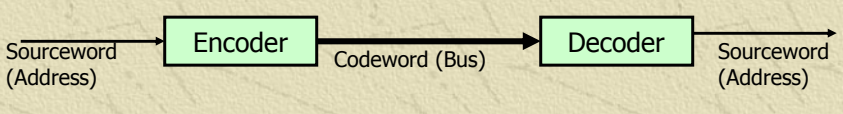
Memory Bus

- 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

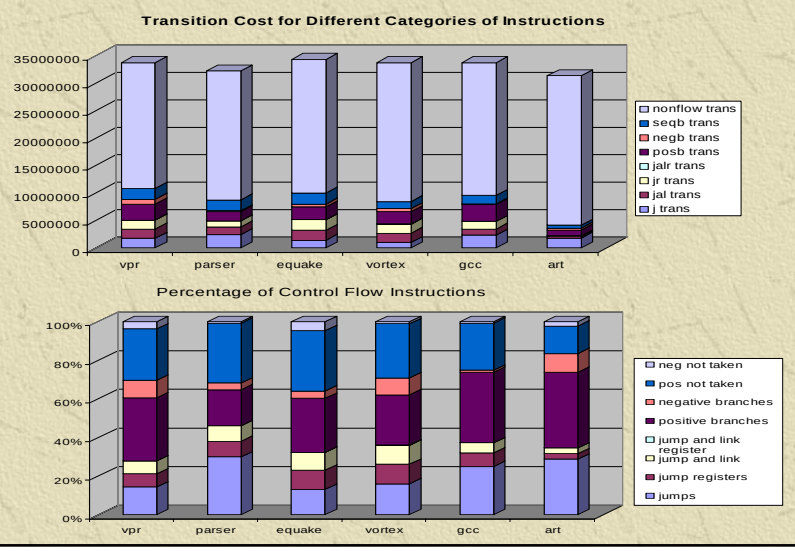
Guidelines

- 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

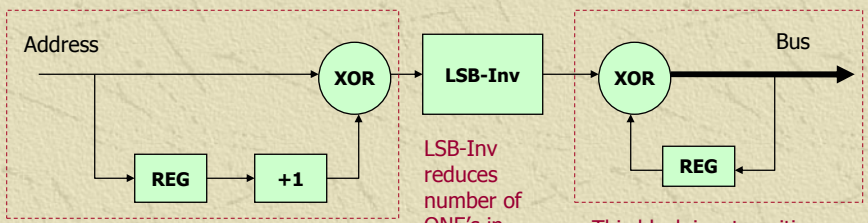
Why SimpleScalar

- 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



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 reduces number of ONE's in small negative numbers

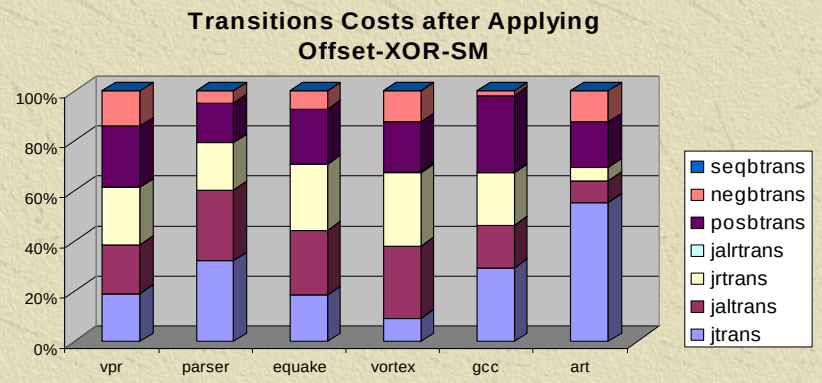
This block is a transition signaling block and sends the values by XORing them with the previous value of the bus

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

Offset	LSB-Inv(Offset)
0000 0101, (+5)	0000 0101
FFFF FFFF, (–1)	8000 0000
FFFF FFFB, (–5)	8000 0004
8000 0000	FFFF FFFF

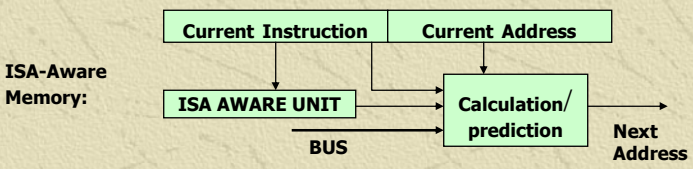
Transition Costs after Offset-XOR-SM

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



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

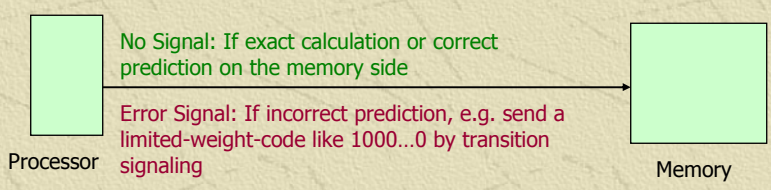


Control Flow Instructions

- *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

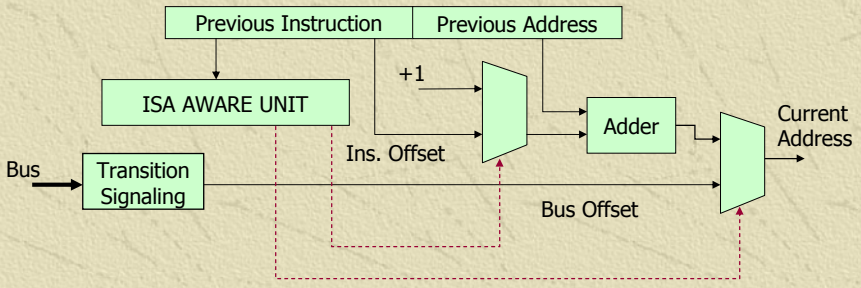
Branch Prediction

- 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



Memory Decoder

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



Results of Predicting BRANCH, J and JAL

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

- 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:

...
404008
46d10a
...
...

In memory:

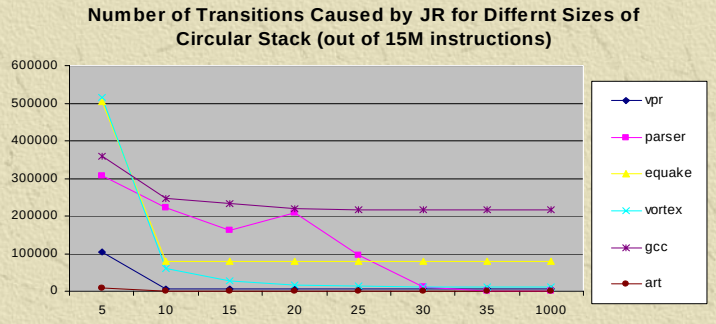
```
...
JR Link_Register
...
...
...
JR Link_Register
```

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



Results

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

- 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.

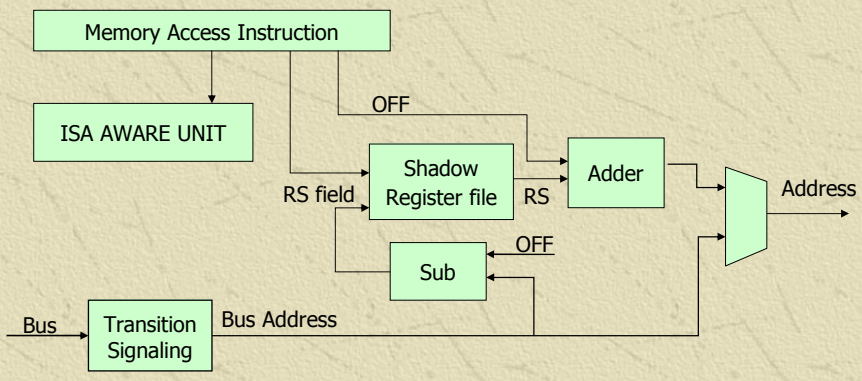
  MEM-INS Rt, (Rs)OFF

  For example for load instruction: MEM(Rs+OFF) → Rt

MEM-INS	Rt Field	Rs Field	OFF

  - 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 *Rs* that has been employed in that instruction)
- If *Rs* does not change between two consecutive memory accesses, no activity will occur on the bus
- An example for R_8 is shown below

R_8 in processor	R_8 in memory	Instruction	Bus Status	Prediction
100	100	Mem Access using R_8	No Activity	OK
100→80	100	Modifies R_8	No bus access	
80	100→80	Mem Access using R_8	Activity	FAIL
80	80	Mem Access using R_8	No Activity	OK

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
...
...
Mem_Ins R14, R10(-16)   <-- Prediction OK
...
Mem_Ins R14, R6(-16)    <-- Prediction Fail, R10 is evicted and R6 is stored instead
...
...
ADD R6, R13,R7          <-- R6 is invalidated
```

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

	Instruction Codec	Data Codec
Num of Literals	686	720
Area (* 1000 λ^2)	343	588
Num of Gates	311	528
Original Bus Power (uW)	5205	20050
Bus Power with BEAM (uW)	416	6055
Power of BEAM memory Codec (uW)	364	1144
Codec power + Bus Power with BEAM	780	7199
Power Saved with BEAM (uW)	4421	12851
Percentage Saving over Bus	85%	64%

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