Very Long Instruction Word Architecture and the ELI-512

Joseph A. Fisher

Yale University

ISCA 1983

Presenter: Alexander Eichhorn

25.11.2021

Executive Summary

- **Motivation**
  - Processors don’t make full use of independent instructions
- **Idea**
  - Have a Very Long Instruction Word with multiple instructions in one
  - Build a compiler to schedule independent instruction in parallel using Trace Scheduling, Loop Unrolling, Memory Bank Prediction
  - Build a hardware prototype: ELI-512
- **Results**
  - 10-30 speedup for ELI-512

Outline

- Problems & Main Idea of VLIW
- VLIW Compiler
  - Trace Scheduling
  - Memory Anti-Aliasing
  - Jump Mechanisms
  - Memory Bank Prediction
- ELI-512
- Current Use
- Strengths & Weaknesses
- Discussion

Problem

- Speed
- Parallel execution
- Speedup from parallelism never more than 2 or 3

Superscalar

instructions

```
addi $t0, $zero, 3
addi $t0, $t0, -4
add $t1, $t1, $a0
addi $a0, $a0, 1
```

hardware

1. cycle

```
addi $t0, $zero, 3
add $t1, $t1, $a0
```

2. cycle

```
addi $t0, $t0, -4
addi $a0, $a0, 1
```

independent instructions scheduled to run in same cycle at runtime

VLIW

Very Long Instruction Word

previous instructions

```
addi $t0, $zero, 3
addi $t0, $t0, -4
add $t1, $t1, $a0
addi $a0, $a0, 1
```

VLIW compiled instructions

1. instruction

```
addi $t0, $zero, 3      add $t1, $t1, $a0
```

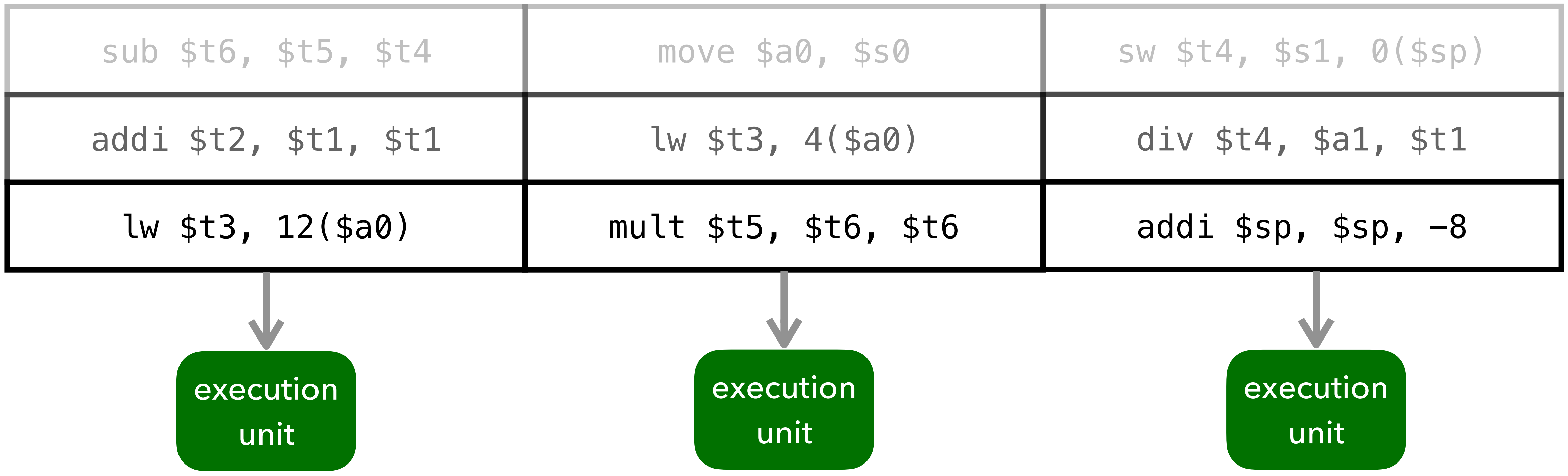
2. instruction

```
addi $t0, $t0, -4       addi $a0, $a0, 1
```

independent instructions scheduled to run in same cycle at compile time

VLIW

Very Long Instruction Word



VLIW vs. Superscalar

Advantage

- Simpler hardware

Disadvantage

- Compiler needs to find independent instructions
- Code size increase in case there's too much dependencies

Problem II: Compiler for VLIW

Simple Calculations

```
x = (a + b) * (a − c)
```

↓

```
x0 = a + b

x1 = a − c

x = x0 * x1
```

↓

`x0 = a + b`	`x1 = a − c`
`x = x0 * x1`	`NOP`

Problem II: Compiler for VLIW

Jumps

	`mult ...`	`xor ...`	`and ...`	`div ...`
fetch	`add ...`	`lw ...`	`lw ...`	`sw ...`
decode	`slt ...`	`mult ...`	`or ...`	`addi ...`
execute	`or ...`	`add ...`	`lw ...`	**`beq ...`**
writeback				

Problem II: Compiler for VLIW

Jumps

N slots

fetch	`add ...`	`lw ...`	`lw ...`	`sw ...`
decode	`slt ...`	`mult ...`	`or ...`	`addi ...`
execute	`or ...`	`add ...`	`lw ...`	**`beq ...`**

On misprediction of a branch, if we're getting a 3 cycle penalty, we actually have to flush 3N (here 12) instructions

Outline

- Problems & Main Idea of VLIW
- VLIW Compiler
  - **Trace Scheduling**
  - Memory Anti-Aliasing
  - Jump Mechanisms
  - Memory Bank Prediction
- ELI-512
- Current Use
- Strengths & Weaknesses
- Discussion

Compiler for VLIW

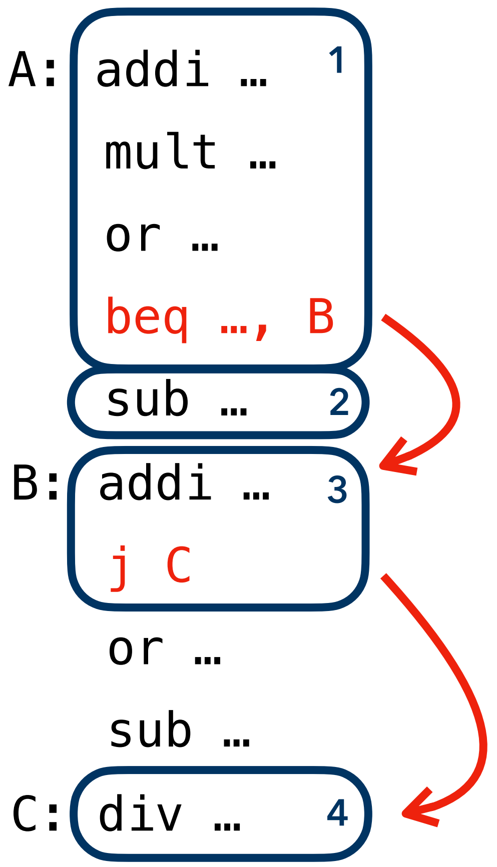
Trace Scheduling

- Trace Scheduling: A Technique for Global Microcode Compaction
  [J. Fisher, IEEE 1981]
- Divide code up into **basic blocks**
    - **Basic block** is a block of code that has no jumps in except at the beginning and no jumps out except at the end

Compiler for VLIW

Trace Scheduling

Basic block is a block of code that has no jumps in except at the beginning and no jumps out except at the end



```
A: addi …      1
   mult …
   or …
   beq …, B
   sub …       2
B: addi …      3
   j C
   or …
   sub …
C: div …       4
```

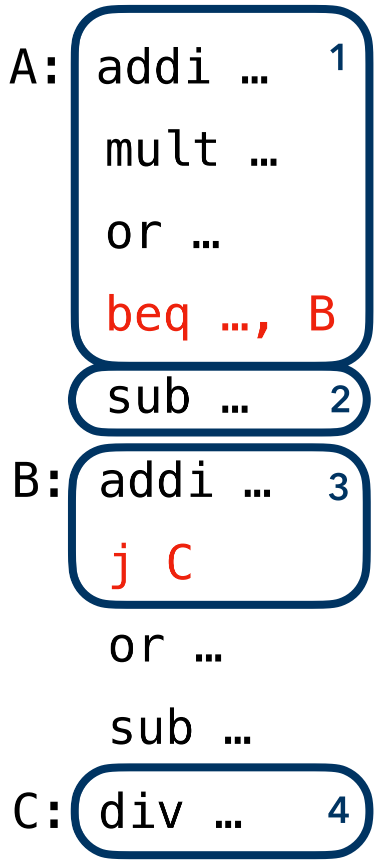
$$\begin{tikzcd}
 & \boxed{1} \arrow[dl, bend right] \arrow[dd] \\
\boxed{2} \arrow[dr, bend right] & \\
 & \boxed{3} \arrow[d] \\
 & \boxed{4}
\end{tikzcd}$$

Compiler for VLIW

Trace Scheduling

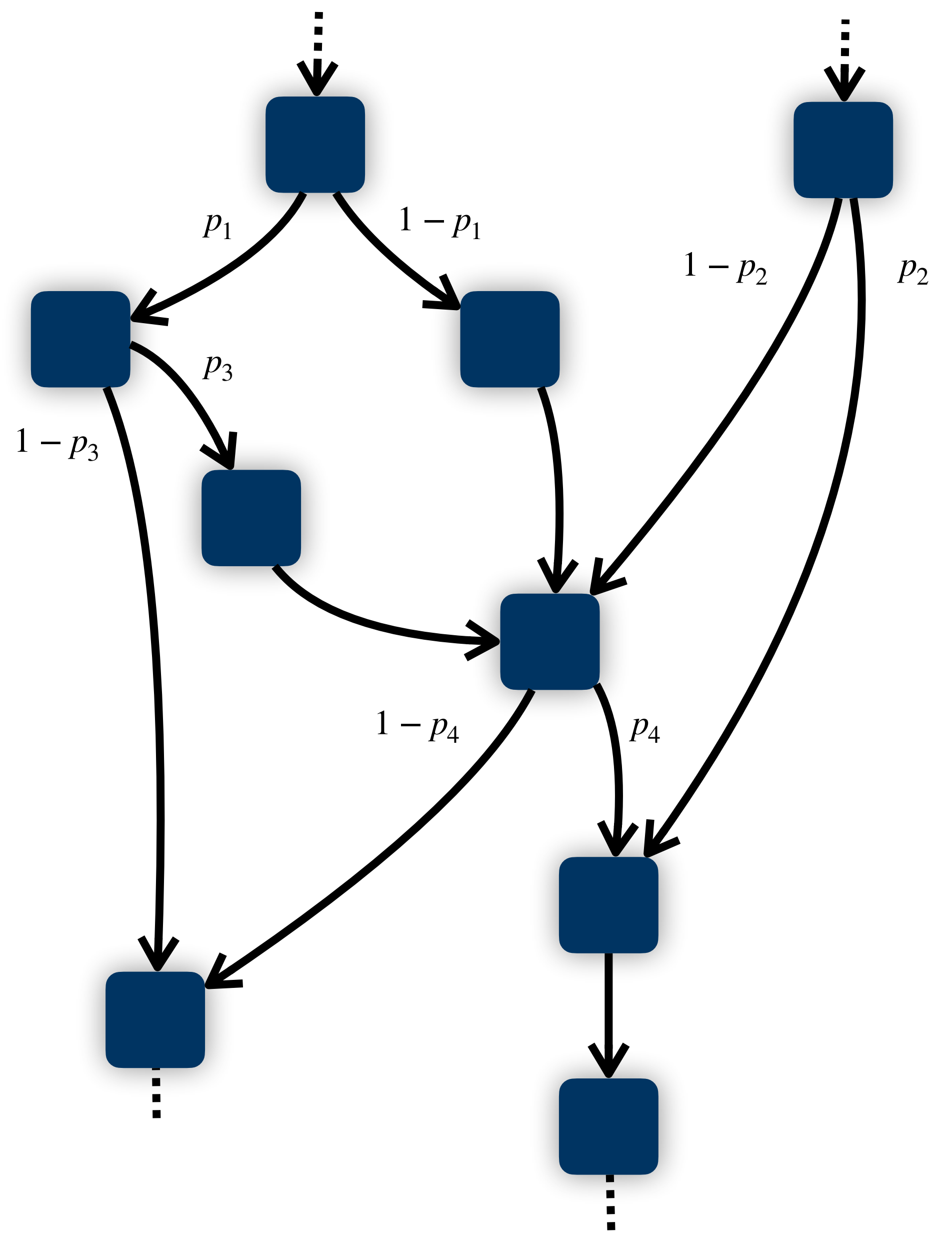
Property of basic blocks

- No outside influence: registers/memory only changed by instructions inside block
- Easy to find independent instructions inside basic block
    - Rearrange independent instructions (traditionally)
    - Schedule independent instructions in same cycle (VLIW)



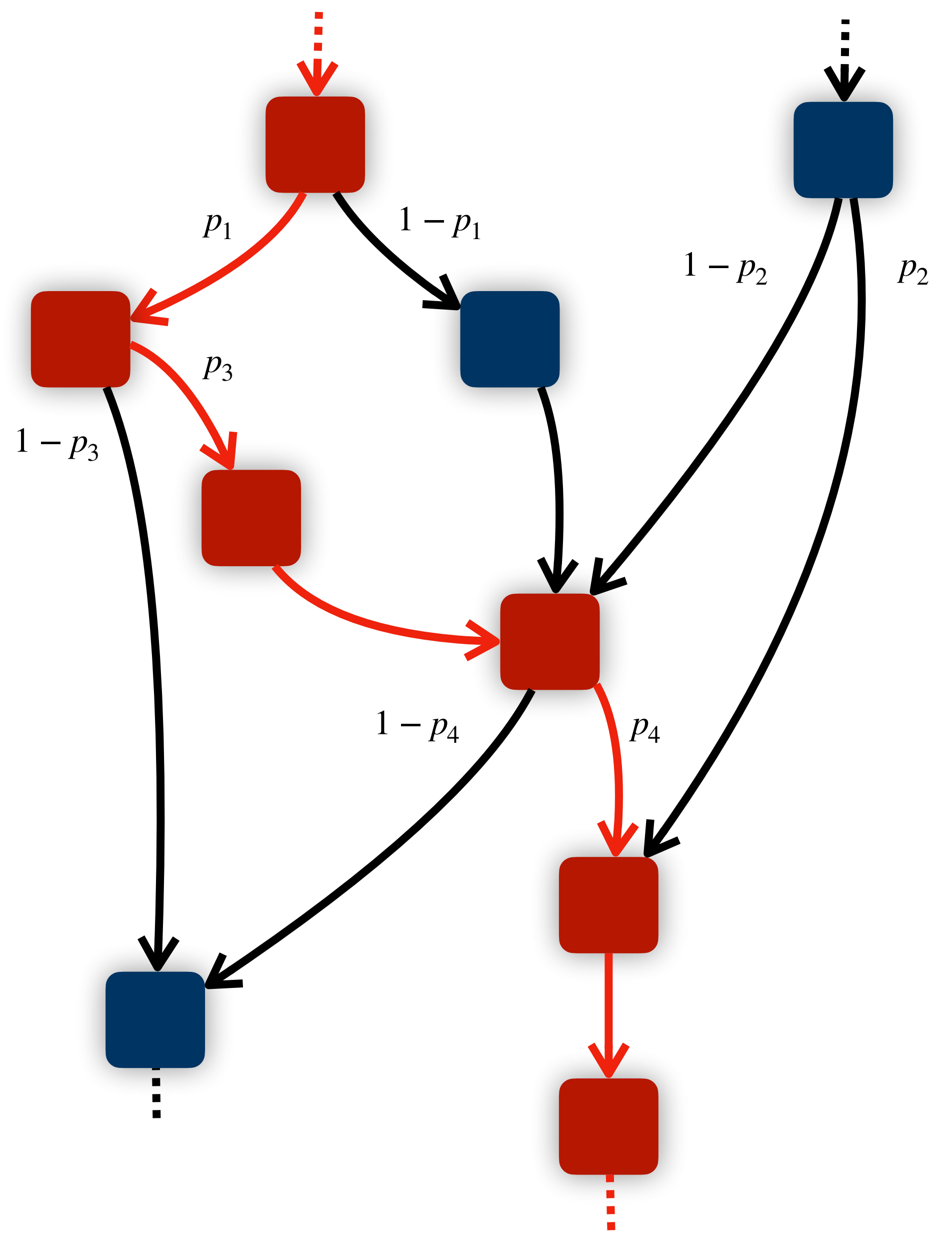
Compiler for VLIW

Trace Scheduling



Compiler for VLIW

Trace Scheduling

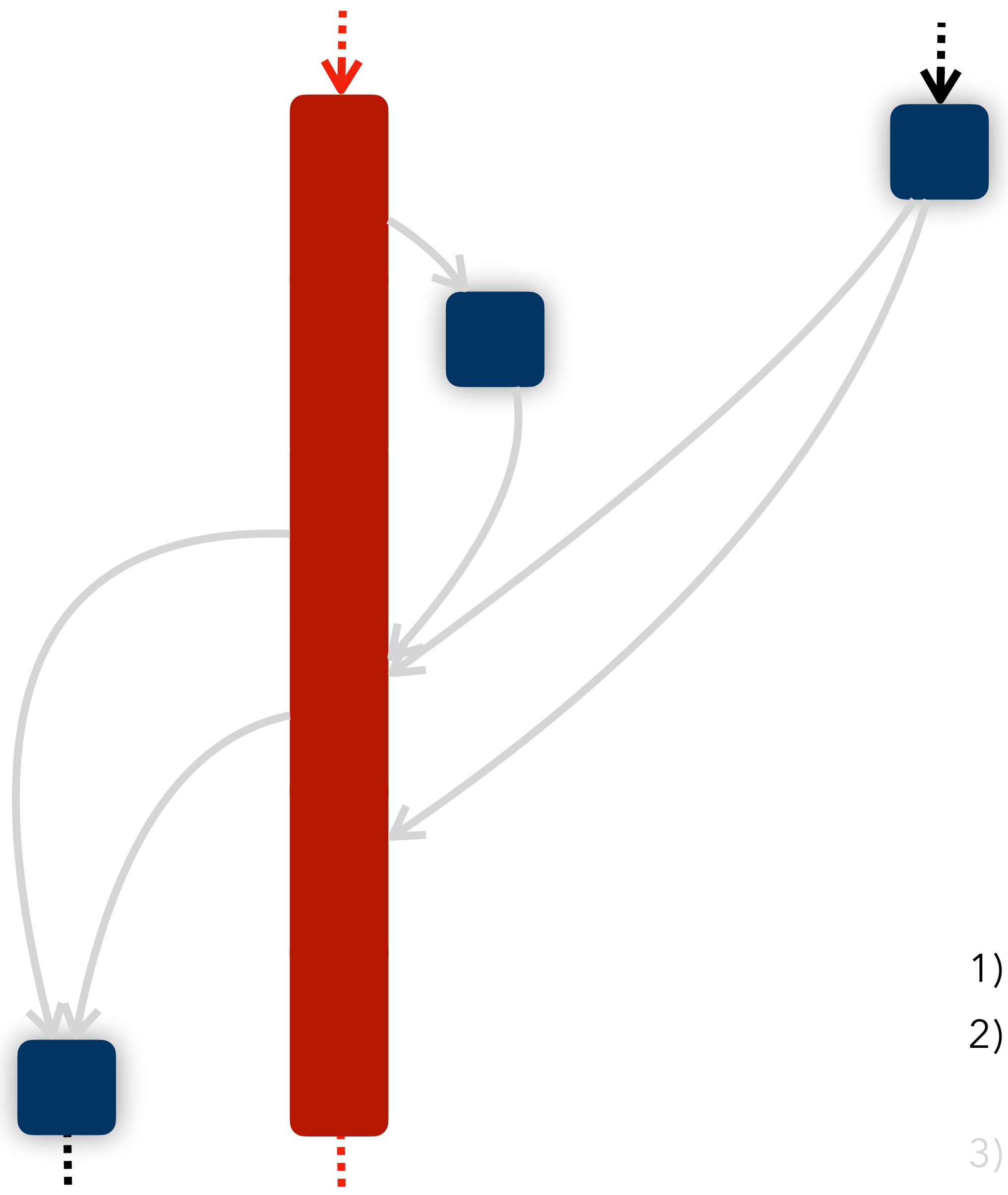


if $p_1, p_3, p_4 > 0.5$

1) determine most likely trace
2) combine trace into one basic block & reschedule code inside
3) add fix-up code to guarantee expected state when jumping out/in
4) recurse on not yet optimized code

Compiler for VLIW

Trace Scheduling



1) determine most likely trace

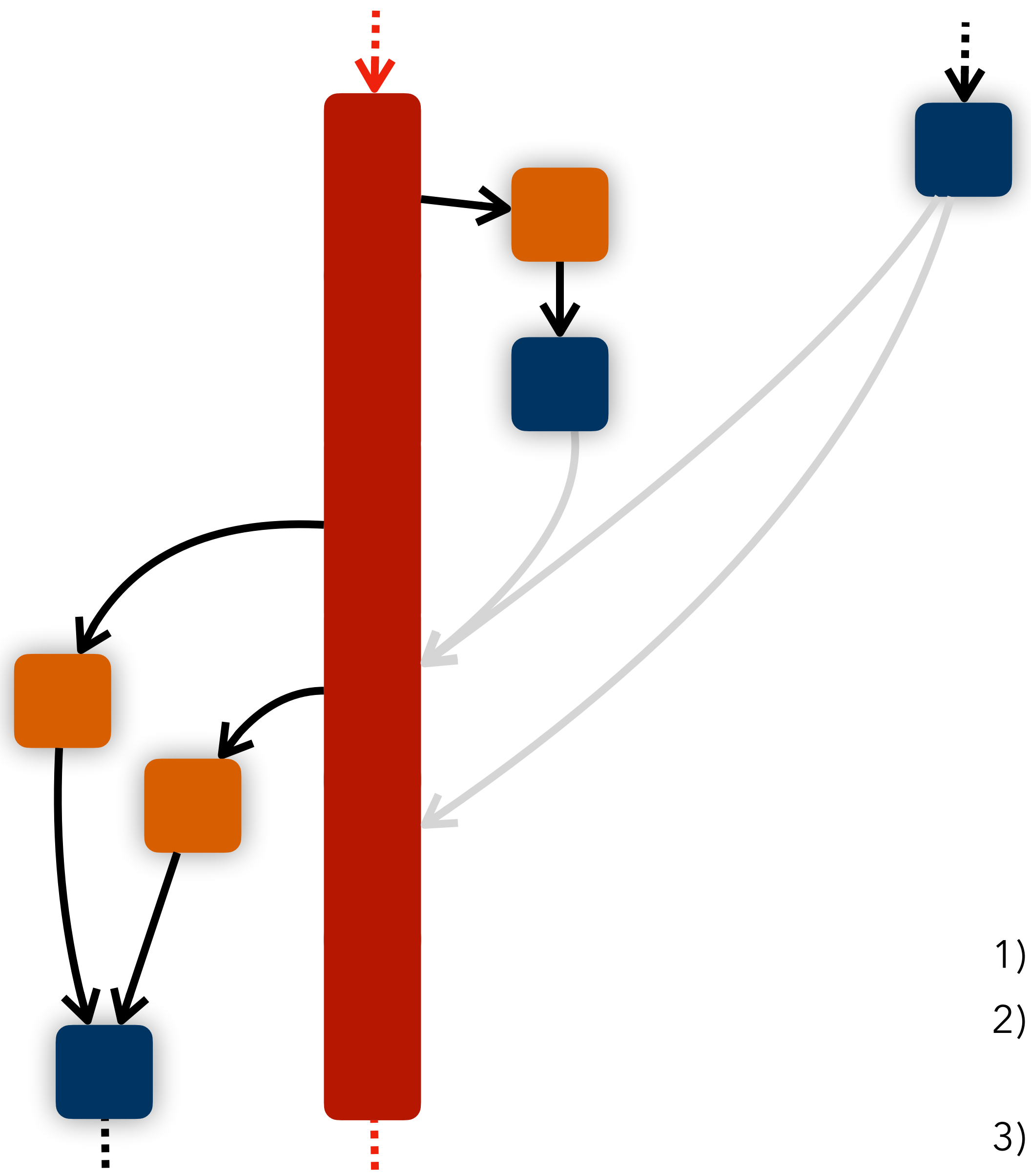
2) combine trace into one basic block & reschedule code inside

3) add fix-up code to guarantee expected state when jumping out/in

4) recurse on not yet optimized code

Compiler for VLIW

Trace Scheduling



1) determine most likely trace

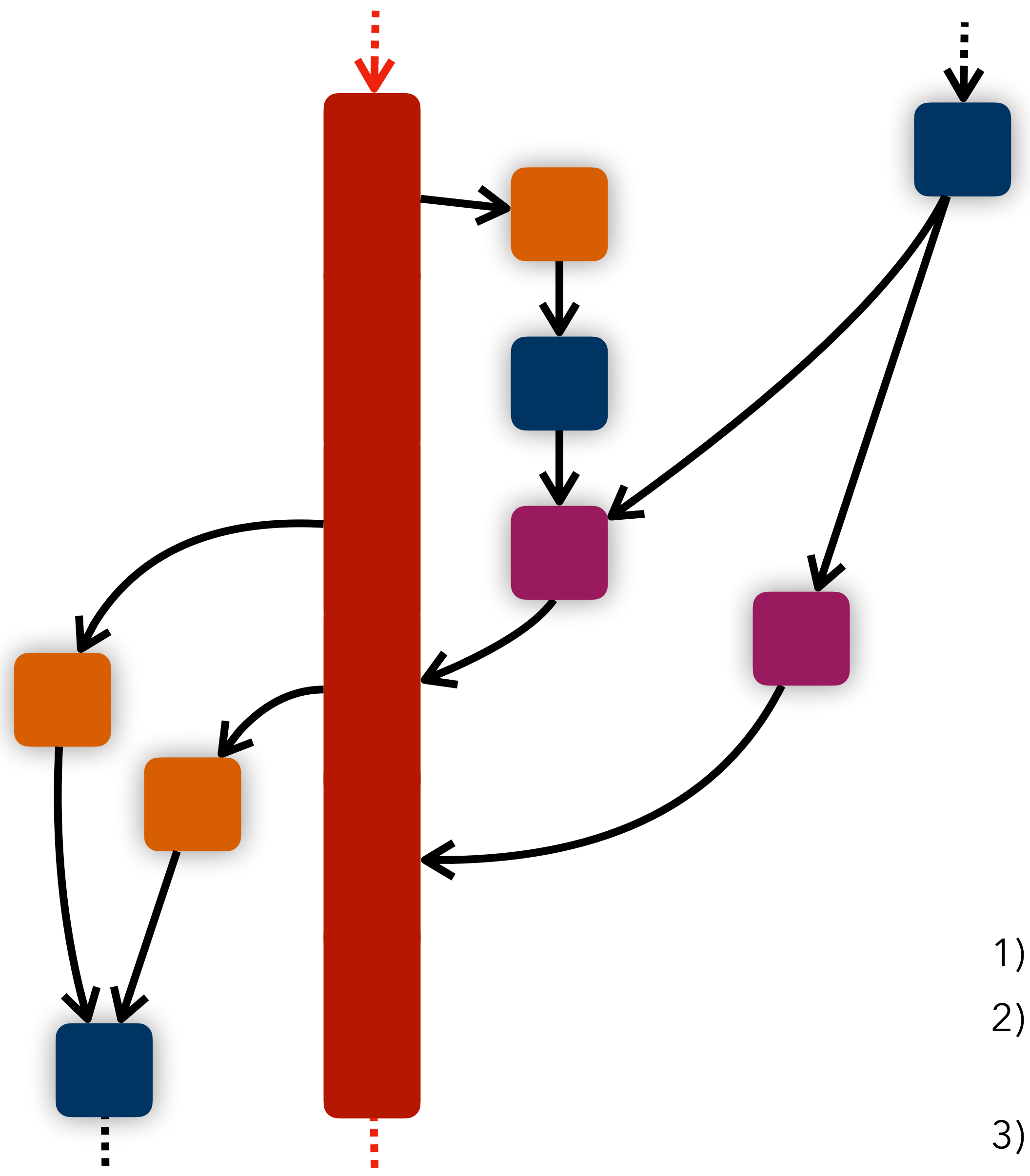
2) combine trace into one basic block & reschedule code inside

3) add fix-up code to guarantee expected state when jumping out/in

4) recurse on not yet optimized code

Compiler for VLIW

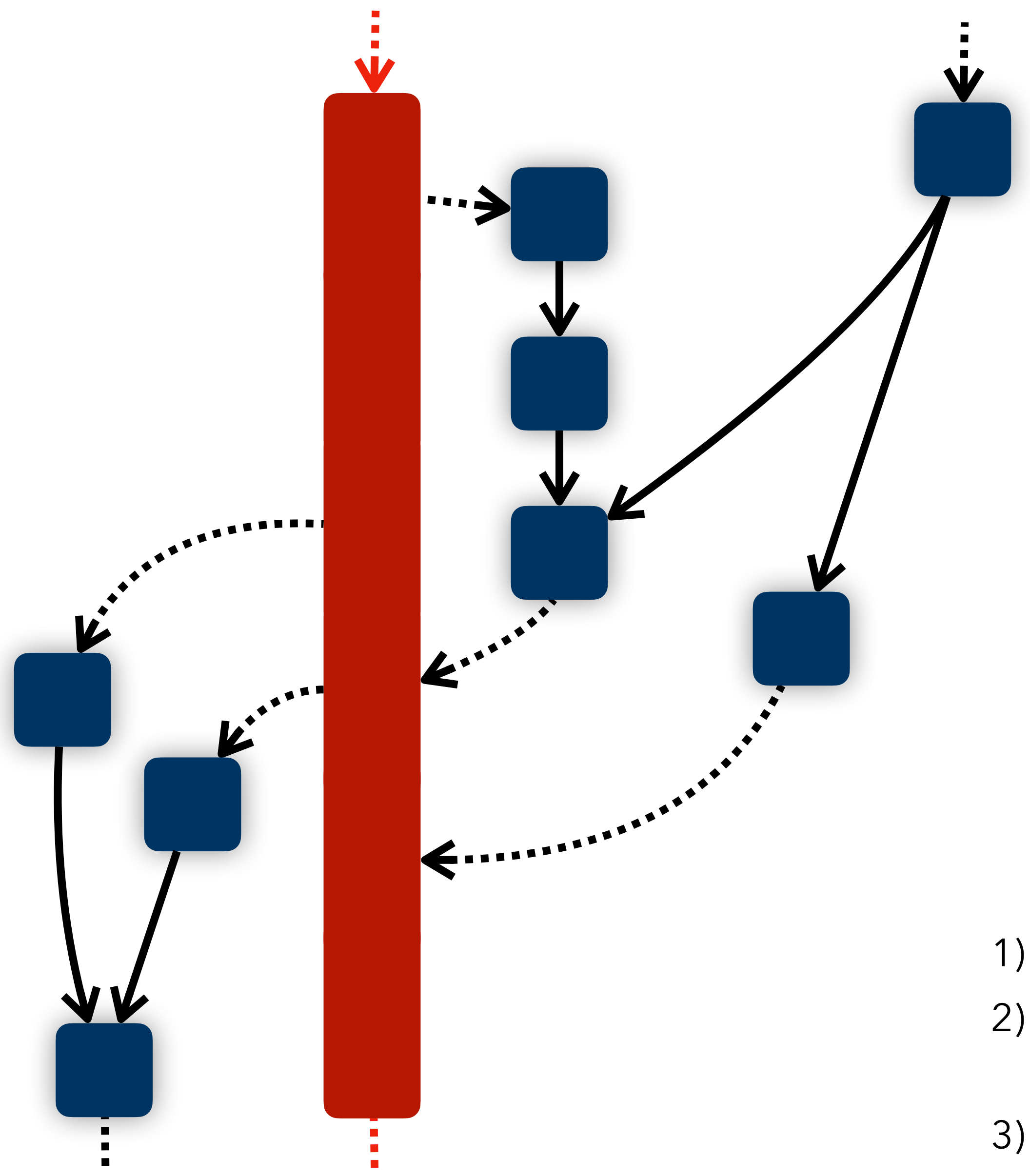
Trace Scheduling



1) determine most likely trace
2) combine trace into one basic block & reschedule code inside
3) add fix-up code to guarantee expected state when jumping out/in
4) recurse on not yet optimized code

Compiler for VLIW

Trace Scheduling



1) determine most likely trace
2) combine trace into one basic block & reschedule code inside
3) add fix-up code to guarantee expected state when jumping out/in
4) recurse on not yet optimized code

Compiler for VLIW

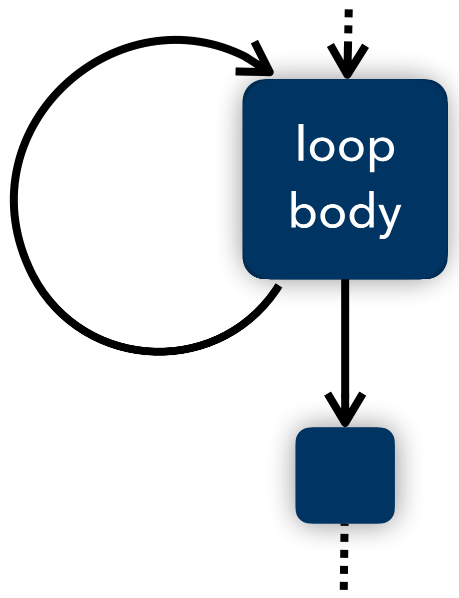
Trace Scheduling

Loops

- Supported by trace scheduling
- However: can be optimized using **loop unrolling**
  - by compiler instead of programmer

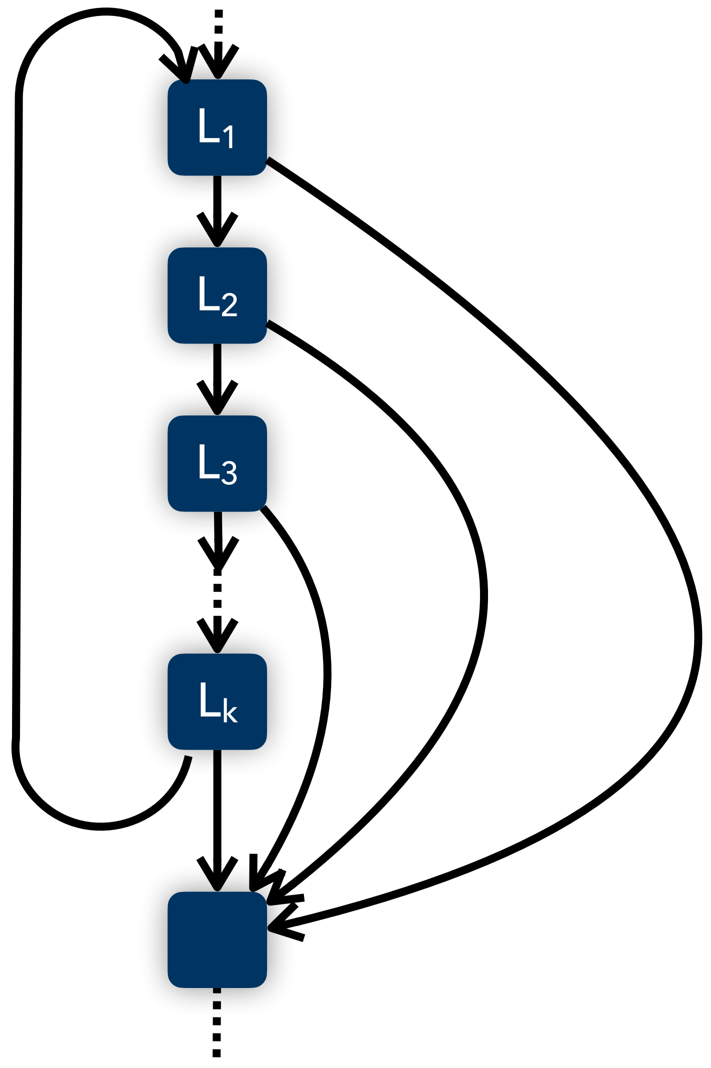
Compiler for VLIW

Trace Scheduling



Compiler for VLIW

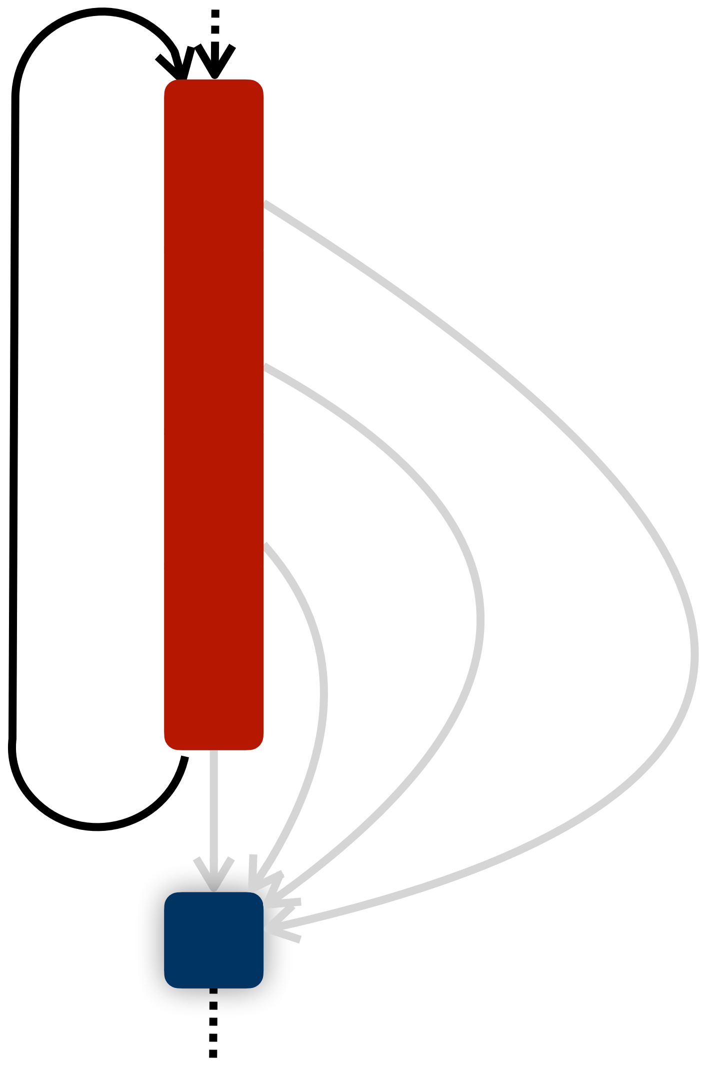
Trace Scheduling



unrolled **k** times

Compiler for VLIW

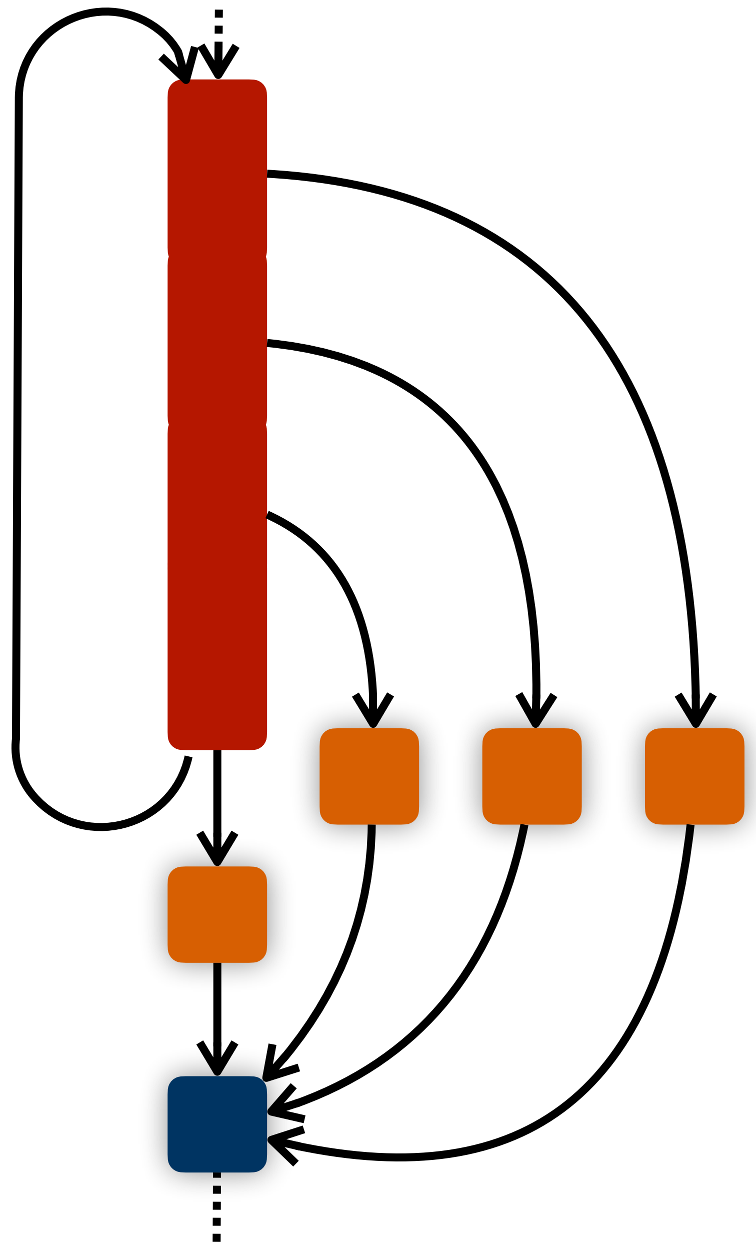
Trace Scheduling



unrolled **k** times → combined into one block to optimize

Compiler for VLIW

Trace Scheduling



Outline

- Problems & Main Idea of VLIW
- VLIW Compiler
    - Trace Scheduling
    - **Memory Anti-Aliasing**
    - Jump Mechanisms
    - Memory Bank Prediction
- ELI-512
- Current Use
- Strengths & Weaknesses
- Discussion

Compiler for VLIW

Memory Anti-Aliasing

```
Z = A + X
A = Y + W
```

can’t reschedule code lines

Compiler for VLIW

Memory Anti-Aliasing

```
Z = A[expr1] + X
A[expr2] = Y + W
```

- If A is an array, the compiler tries to solve the equation `expr1 = expr2`
- Assume `expr1` and `expr2` are integers
- Use diophantine equation solver
  - no solution = `expr1` and `expr2` are not the same

    → can be rescheduled

Compiler for VLIW

Memory Anti-Aliasing

Example

```
for (i = k to n) {
    Z = A[i*i] + X
    A[i+1] = Y + W
    ...
}
```

- Solve equation: $i^2 = i + 1$
- There exist non-integer solutions: $\frac{1}{2} \pm \frac{\sqrt{5}}{2}$
- But no integer solutions exist

  → `A[i*i]` and `A[i+1]` never access the same element

Outline

- Problems & Main Idea of VLIW
- VLIW Compiler
  - Trace Scheduling
  - Memory Anti-Aliasing
  - **Jump Mechanisms**
  - Memory Bank Prediction
- ELI-512
- Current Use
- Strengths & Weaknesses
- Discussion

Compiler for VLIW

Jump Mechanisms

`beq $t0, $t1, A`	`beq $t2, $t3, B`	`beq $t4, $t5, C`	`beq $t6, $t7, D`

`j A`	`j B`	`beq $t4, $t5, C`	`beq $t6, $t7, D`

→ leads to ambiguity when executed

Compiler for VLIW

Jump Mechanisms

Possibilities

- Only allow 1 jump/branch instruction per cycle
- Multiple branch instructions with priority rules

Compiler for VLIW

Jump Mechanisms

Solution

- per cycle:
  - n independent tests
  - n+1 location to jump to
- Earlier tests have priority

```
COND (test1 label1)
     (test2 label2)
         ⋮
     (testn labeln)
     (TRUE label-fall-through)
```

C-style implementation

```
if (test1) goto label1
if (test2) goto label2
        ⋮
if (testn) goto labeln
goto label-fall-through
```

Compiler for VLIW

Jump Mechanisms

Encoding Size

- Placing n+1 full address candidates in instruction
  - Uses a lot of instruction bits

- Conform to certain regularities

Compiler for VLIW

Jump Mechanisms

n = 3

NEXT INSTRUCTION ADDRESS: specified in each instruction

Test Field	Condition	Potential Jump Address
0	test1	00 [NEXT INSTRUCTION ADDRESS]
1	test2	01 [NEXT INSTRUCTION ADDRESS]
2	test3	10 [NEXT INSTRUCTION ADDRESS]
3	TRUE	11 [NEXT INSTRUCTION ADDRESS]

Compiler for VLIW

Jump Mechanisms

n = 3

NEXT INSTRUCTION ADDRESS: 00000011

Test Field	Condition	Potential Jump Address
0	test1	00 00000011
1	test2	01 00000011
2	test3	10 00000011
3	TRUE	11 00000011

Compiler for VLIW

Jump Mechanisms

single branch with **test1** can be encoded like this:

Test Field	**Condition**	**Potential Jump Address**
0	test1	00 00000011
1	TRUE	01 00000011
2	-	10 00000011
3	TRUE	11 00000011

Compiler for VLIW

Jump Mechanisms

or **jump** to 10 11010111:

Test Field	**Condition**	**Potential Jump Address**
0	FALSE	**00** 11010111
1	FALSE	**01** 11010111
2	TRUE	**10** 11010111
3	TRUE	**11** 11010111

Outline

- Problems & Main Idea of VLIW
- VLIW Compiler
    - Trace Scheduling
    - Memory Anti-Aliasing
    - Jump Mechanisms
    - **Memory Bank Prediction**
- ELI-512
- Current Use
- Strengths & Weaknesses
- Discussion

Compiler for VLIW

Memory Bank Prediction

Problem

- Multiple memory references in same cycle
  - Requires global arbitration system

    → latency increase
  - Possible memory bank conflict

    → freezes entire processor

Compiler for VLIW

Memory Bank Prediction

Solution

- Static code
  - Scalars always have known locations
  - Array values can be predicted by the same system that does anti-aliasing
- Loops
  - Often unrolling by a multiple of number of banks allows to predict banks

! **Subscript of the array might have data-dependent starting point**

Compiler for VLIW

Memory Bank Prediction

Solution for data-dependent starting point: **adding a pre-loop**

```
for i = k to n {
    arr[i] += 1
}
```

loop unrolling

```
             i = k
LOOP:        if (i >= n) goto FALLTHROUGH
             arr[i] += 1
             i++
             if (i >= n) goto FALLTHROUGH
             arr[i] += 1
             i++
                 ⋮
             if (i < n) goto LOOP
FALLTHROUGH:
```

add pre-loop

```
             i = k
PRELOOP:     if (i mod #BANKS == 0) goto LOOP
             arr[i] += 1
             i++
             if (i > n) goto FALLTHROUGH
             goto PRELOOP

LOOP:        if (i >= n) goto FALLTHROUGH
             arr[i] += 1
             i++
             if (i >= n) goto FALLTHROUGH
             arr[i] += 1
             i++
                 ⋮
             if (i < n) goto LOOP
FALLTHROUGH:
```

Outline

- Problems & Main Idea of VLIW
- VLIW Compiler
    - Trace Scheduling
    - Memory Anti-Aliasing
    - Jump Mechanisms
    - Memory Bank Prediction
- **ELI-512**
- Current Use
- Strengths & Weaknesses
- Discussion

ELI-512

- ELI = Enormously Longword Instructions
- 512 bit instruction word
- 16 clusters with an ALU and some storage
  - 8 M-clusters and 8 F-clusters

ELI-512

M-clusters

- Integer ALU
- Multiport integer register bank
- Local memory module

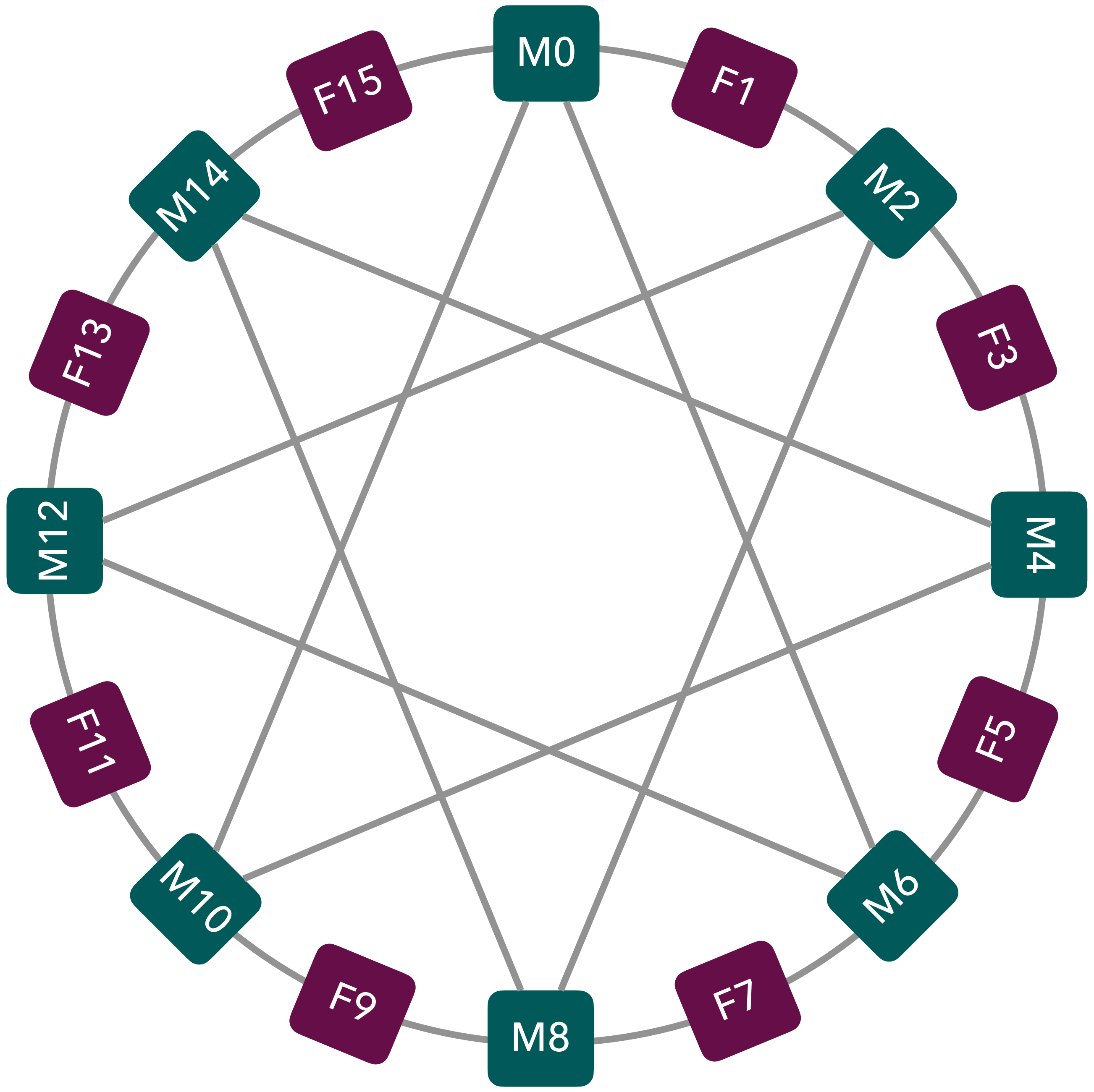
  → directly accessed when bank is known

F-clusters

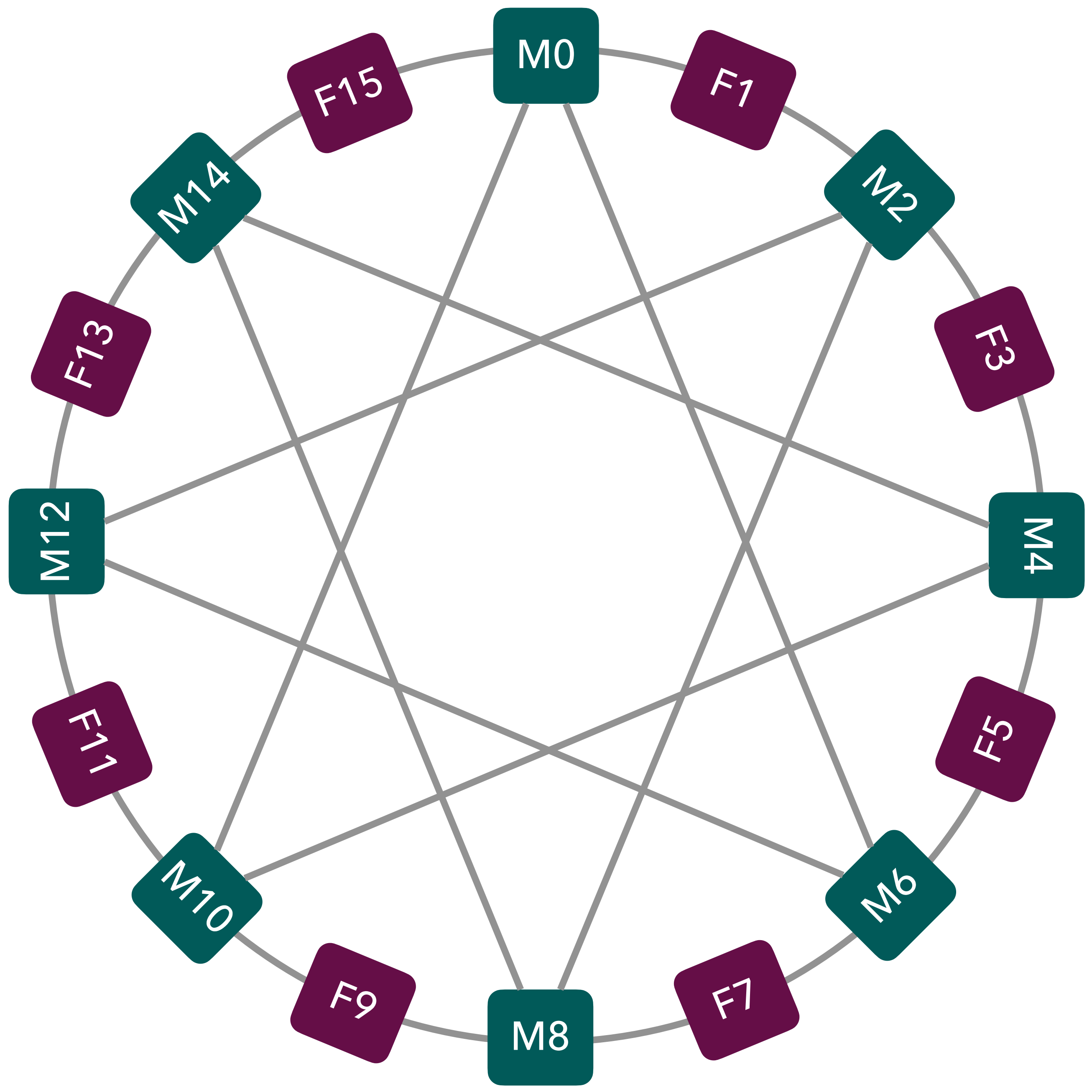
- Floating point ALU
- Multiport floating register bank

- Additionally 2 access ports to address memory globally (bank unknown) → slower

ELI-512



ELI-512



More about how the compiler has to optimize for these connections:

Parallel Processing: A Smart Compiler and a Dumb Machine
[J. Fisher et al., 1984]

ELI-512

Each instruction cycle:

- 16 ALU operations (8 restricted to 32-bit integer)
- 8 pipelined memory references
- 1 multiway conditional jump

Outline

- Problems & Main Idea of VLIW
- VLIW Compiler
  - Trace Scheduling
  - Memory Anti-Aliasing
  - Jump Mechanisms
  - Memory Bank Prediction
- ELI-512
- **Current Use**
- Strengths & Weaknesses
- Discussion

VLIW

Current Technologies for Concurrency

Multicore

- Programmer has to manage concurrency
- No “micro-concurrency”
    - e.g. `x = (a + b) * (a - c)` can’t be efficiently calculated in parallel
- Much more efficient for completely independent programs

VLIW

Current Technologies for Concurrency

SIMD

- Programmer or compiler can manage concurrency
- Optional in CPU
- Obviously only usable for same instruction

VLIW

Commercial Use

- Intel Itanium Architecture (IA-64) on EPIC design concept (1990)
    - Windows XP ported to IA-64
    - but mainly focussed on server market
    - discontinued 2010
- AMD GPU: Radeon HD 2900 with VLIW5 (2007)
    - Radeon HD 6900 with VLIW4
    - discontinued 2011

VLIW

Commercial Use

- Xilinx Versal Chip (2019)
  - AI engine with 6-way VLIW instructions
    - 2x scalar operations
    - 2x memory loads
    - 1x vector multiplication
    - 1x memory store

Outline

- Problems & Main Idea of VLIW
- VLIW Compiler
    - Trace Scheduling
    - Memory Anti-Aliasing
    - Jump Mechanisms
    - Memory Bank Prediction
- ELI-512
- Current Use
- **Strengths & Weaknesses**
- Discussion

Strengths

- No work for programmer to achieve concurrency
- No hardware scheduling
- A lot of solutions for different problems
- Compiler optimization can be used outside VLIW

Weaknesses

- Compiler is computer specific: instruction per cycle, latency of instructions, memory bank
    - recompilation for every change
- If one operation stalls, the whole processor stalls
- Inefficient variable latency operations
    - e.g. memory access with cache
- Code size increase
- No clear results / background of specified speedups not really clear
- ELI-512 not discussed in detail

Discussion

Why was VLIW never really commercially successful?

Discussion

For which today's applications might VLIW be useful?

or which emerging technologies?

Discussion

Adding an extra VLIW core to multicore processors today.
Good idea? Challenges?