

15-213

Code Optimization II

October 2, 2001

Topics

- **Machine-Dependent Optimizations**
 - Pointer code
 - Unrolling
 - Enabling instruction level parallelism
- **Understanding Processor Operation**
 - Translation of instructions into operations
 - Out-of-order execution of operations
- **Branches and Branch Prediction**
- **Advice**

Previous Best Combining Code

```
void combine4(vec_ptr v, int *dest)
{
    int i;
    int length = vec_length(v);
    int *data = get_vec_start(v);
    int sum = 0;
    for (i = 0; i < length; i++)
        sum += data[i];
    *dest = sum;
}
```

Task

- Compute sum of all elements in vector
- Vector represented by C-style abstract data type

General Forms of Combining

```
void abstract_combine4(vec_ptr v, data_t *dest)
{
    int i;
    int length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP data[i];
    *dest = t;
}
```

Data Types

- Use different declarations for `data_t`
- `int`
- `float`
- `double`

class11.ppt

Operations

- Use different definitions of OP and IDENT
- `+` / `0`
- `*` / `1`

Machine Independent Opt. Results

Optimizations

- Reduce function calls and memory references within loop

Method	Integer		Floating Point	
	+	*	+	*
Abstract -g	42.06	41.86	41.44	160.00
Abstract -O2	31.25	33.25	31.25	143.00
Move vec_length	20.66	21.25	21.15	135.00
data access	6.00	9.00	8.00	117.00
Accum. in temp	2.00	4.00	3.00	5.00



Performance Anomaly

- Computing FP product of all elements exceptionally slow.
- Very large speedup when accumulate in temporary
- Caused by quirk of IA32 floating point
 - Memory uses 64-bit format, register use 80
 - Benchmark data caused overflow of 64 bits, but not 80

Pointer Code

```
void combine4p(vec_ptr v, int *dest)
{
    int length = vec_length(v);
    int *data = get_vec_start(v);
    int *dend = data+length;
    int sum = 0;
    while (data < dend) {
        sum += *data;
        data++;
    }
    *dest = sum;
}
```

Optimization

- Use pointers rather than array references
- CPE: 3.00 (Compiled -O2)

–Oops! We're not making progress here!

Warning: Some compilers do better job optimizing array code

Pointer vs. Array Code Inner Loops

Array Code

```
.L24:          # Loop:
    addl  (%eax,%edx,4),%ecx  # sum += data[i]
    incl %edx                # i++
    cmpl %esi,%edx           # i:length
    jl   .L24                # if < goto Loop
```

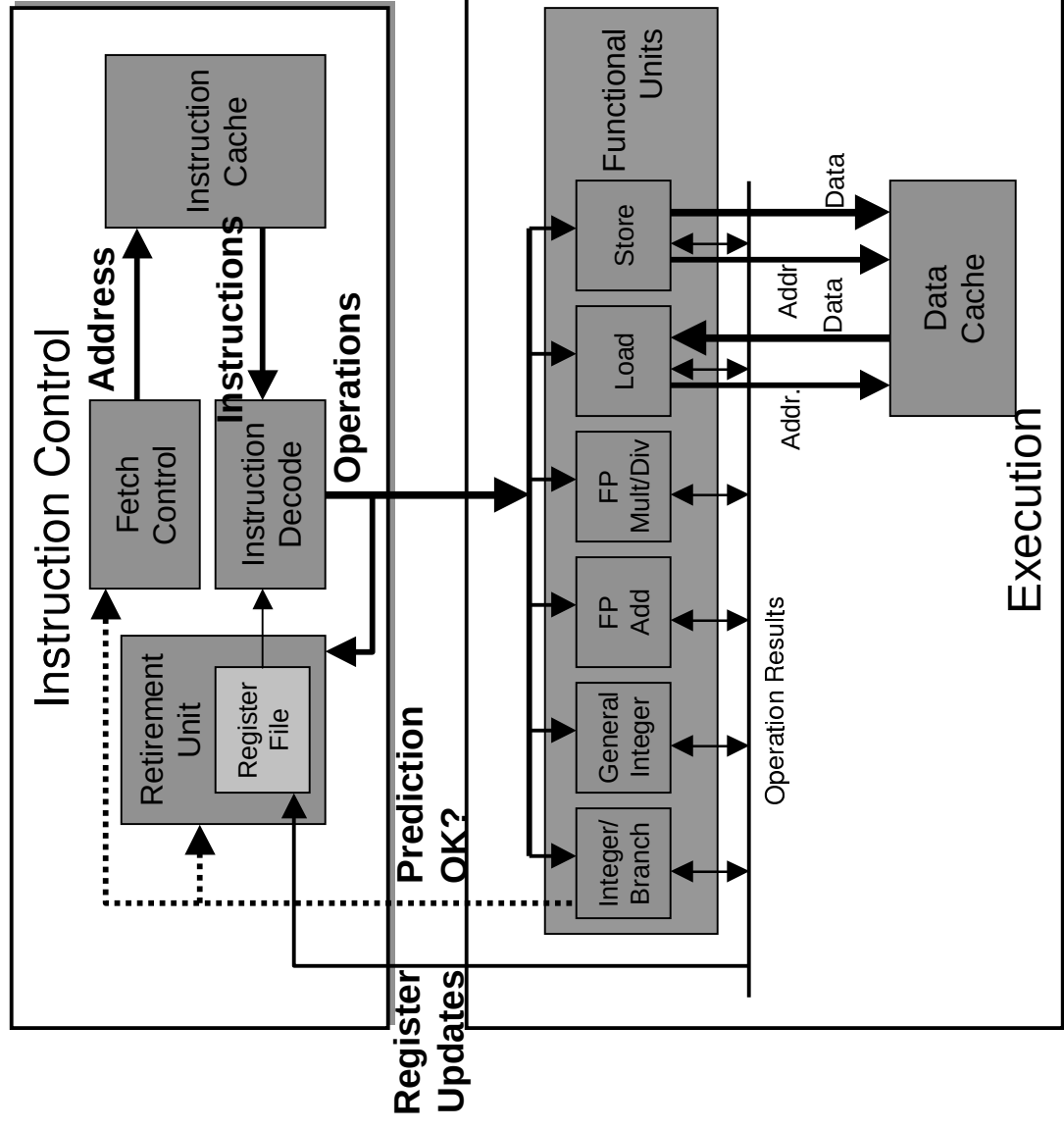
Pointer Code

```
.L30:          # Loop:
    addl  (%eax),%ecx        # sum += *data
    addl  $4,%eax            # data ++
    cmpl  %edx,%eax          # data:dend
    jb   .L30                # if < goto Loop
```

Performance

- Array Code: 4 instructions in 2 clock cycles
- Pointer Code: Almost same 4 instructions in 3 clock cycles

Modern CPU Design



CPU Capabilities of Pentium III

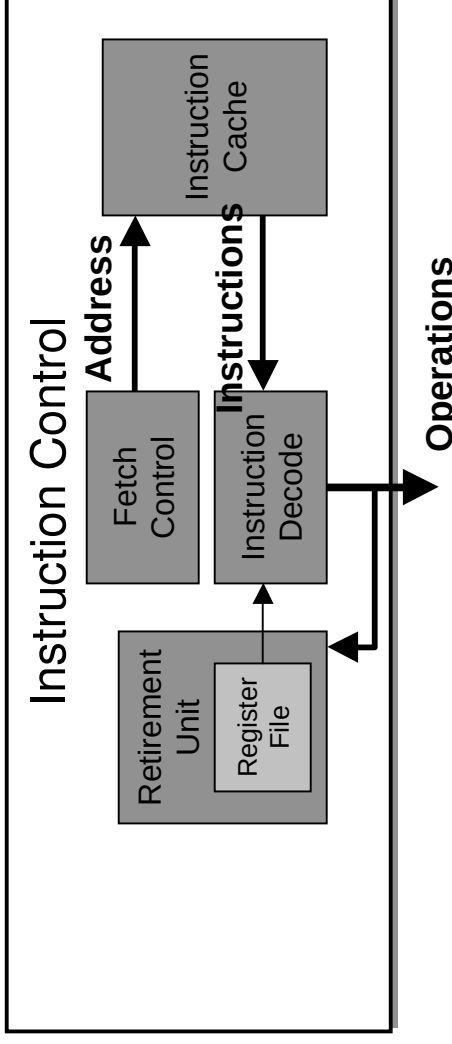
Multiple Instructions Can Execute in Parallel

- 1 load
- 1 store
- 2 integer (one may be branch)
- 1 FP Addition
- 1 FP Multiplication or Division

Some Instructions Take > 1 Cycle, but Can be Pipelined

Instruction	Latency	Cycles/Issue
• Load / Store	3	1
• Integer Multiply	4	1
• Integer Divide	36	36
• Double/Single FP Multiply	5	2
• Double/Single FP Add	3	1
• Double/Single FP Divide	38	38

Instruction Control



Grabs Instruction Bytes From Memory

- Based on Current PC + Predicted Targets for Predicted Branches
- Hardware dynamically guesses whether branches taken/not taken and (possibly) branch target

Translates Instructions Into *Operations*

- Primitive steps required to perform instruction
- Typical instruction requires 1–3 operations

Converts Register References Into *Tags*

- Abstract identifier linking destination of one operation with sources of later operations

Translation Example

Version of Combine4

- Integer data, multiply operation

```
.L24:           # Loop:
imull (%eax,%edx,4),%ecx  # t *= data[i]
incl %edx           # i++
cmpl %esi,%edx       # i:length
jl .L24            # if < goto Loop
```

Translation of First Iteration

```
.L24:
imull (%eax,%edx,4),%ecx

incl %edx
cmpl %esi,%edx
jl .L24
```

```
load (%eax,%edx.0,4) → t.1
imull t.1, %ecx.0 → %ecx.1
incl %edx.0
cmpl %esi, %edx.1 → cc.1
jl-taken cc.1
```

Understanding Translation Example

```
imull (%eax,%edx,4),%ecx
```

```
load (%eax,%edx.0,4) → t.1  
imull t.1, %ecx.0 → %ecx.1
```

- **Split into two operations**

- Load reads from memory to generate temporary result t.1
- Multiply operation just operates on registers

- **Operands**

- Registers %eax does not change in loop. Values will be retrieved from register file during decoding
- Register %ecx changes on every iteration. Uniquely identify different versions as %ecx.0, %ecx.1, %ecx.2, ...
 - » Register *renaming*
 - » Values passed directly from producer to consumers

```
incl %edx
```

```
incl %edx.0 → %edx.1
```

- **Register %edx changes on each iteration. Rename as %edx.0, %edx.1, %edx.2, ...**

Understanding Translation Ex. (cont)

```
cmp $\ell$  %esi, %edx
```

```
cmp $\ell$  %esi, %edx.1 → cc.1
```

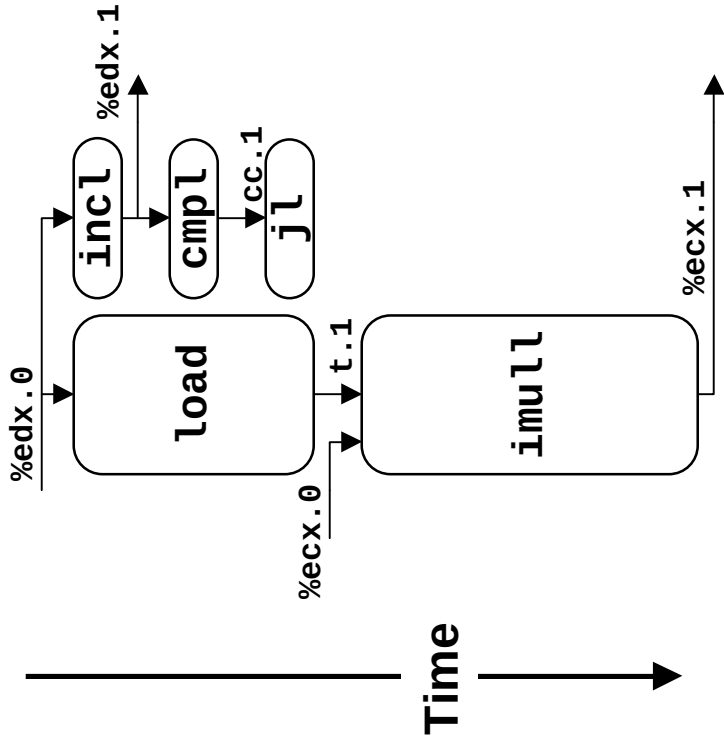
- Condition codes are treated similar to registers
- Assign tag to define connection between producer and consumer

```
j $\ell$  .L24
```

```
j $\ell$ -taken cc.1
```

- Instruction control unit determines destination of jump
- Predicts whether will be taken and target
- Starts fetching instruction at predicted destination
- Execution unit simply checks whether or not prediction was OK
- If not, it signals instruction control
 - Instruction control then “invalidates” any operations generated from misfetched instructions
 - Begins fetching and decoding instructions at correct target

Visualizing Operations



```

load (%eax,%edx,4) → t.1
imull t.1, %ecx.0 → %ecx.1
incl %edx.0
cpl %esi, %edx.1 → cc.1
jl-taken cc.1
    
```

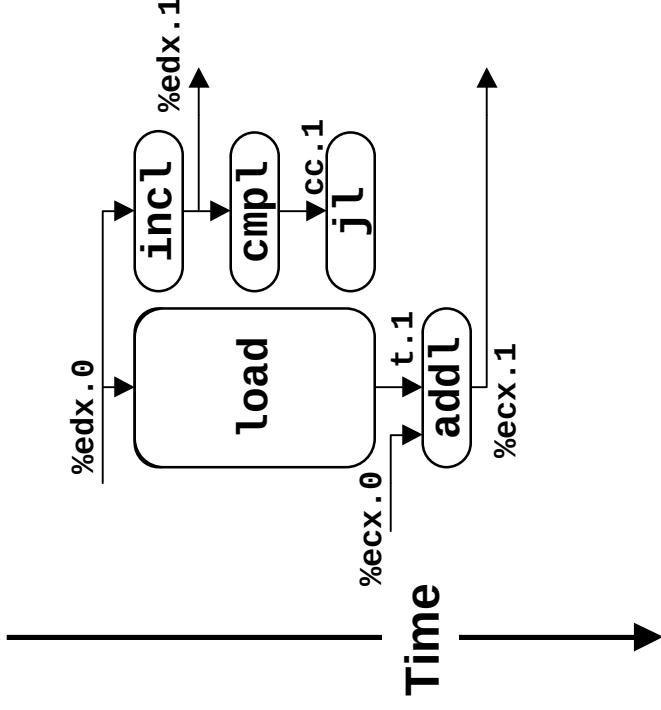
Operations

- Vertical position denotes time at which executed
 - Cannot begin operation until operands available
- Height denotes latency

Operands

- Arcs shown only for operands that are passed within execution unit

Visualizing Operations (cont.)

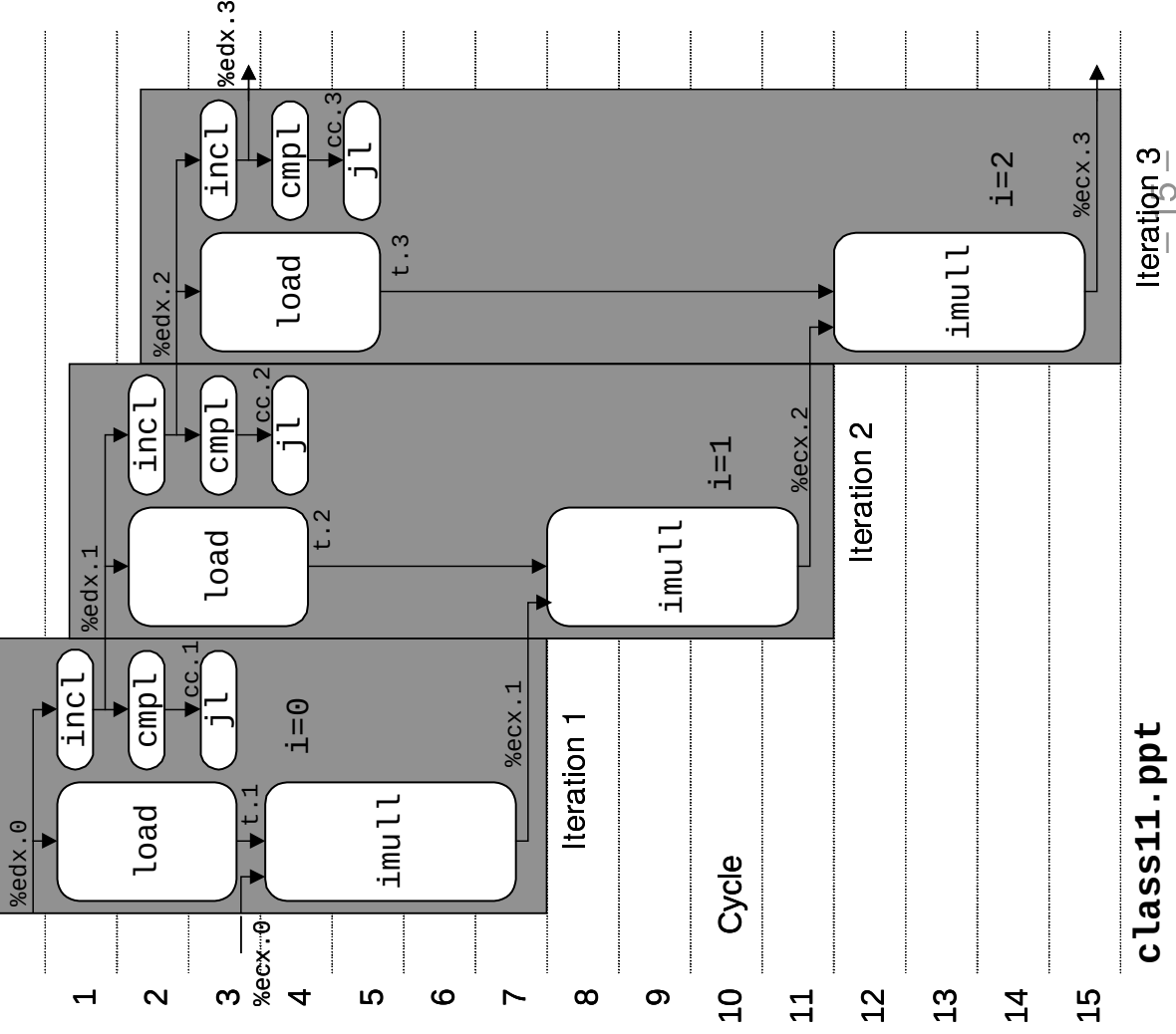


```
load (%eax,%edx,4) → t.1
iaddl t.1, %ecx.0 → %ecx.1
incl %edx.0
cmpl %esi, %edx.1 → cc.1
jnl-taken cc.1
```

Operations

- Same as before, except that add has latency of 1

3 Iterations of Combining Product



class11.ppt

CS 213 F'01

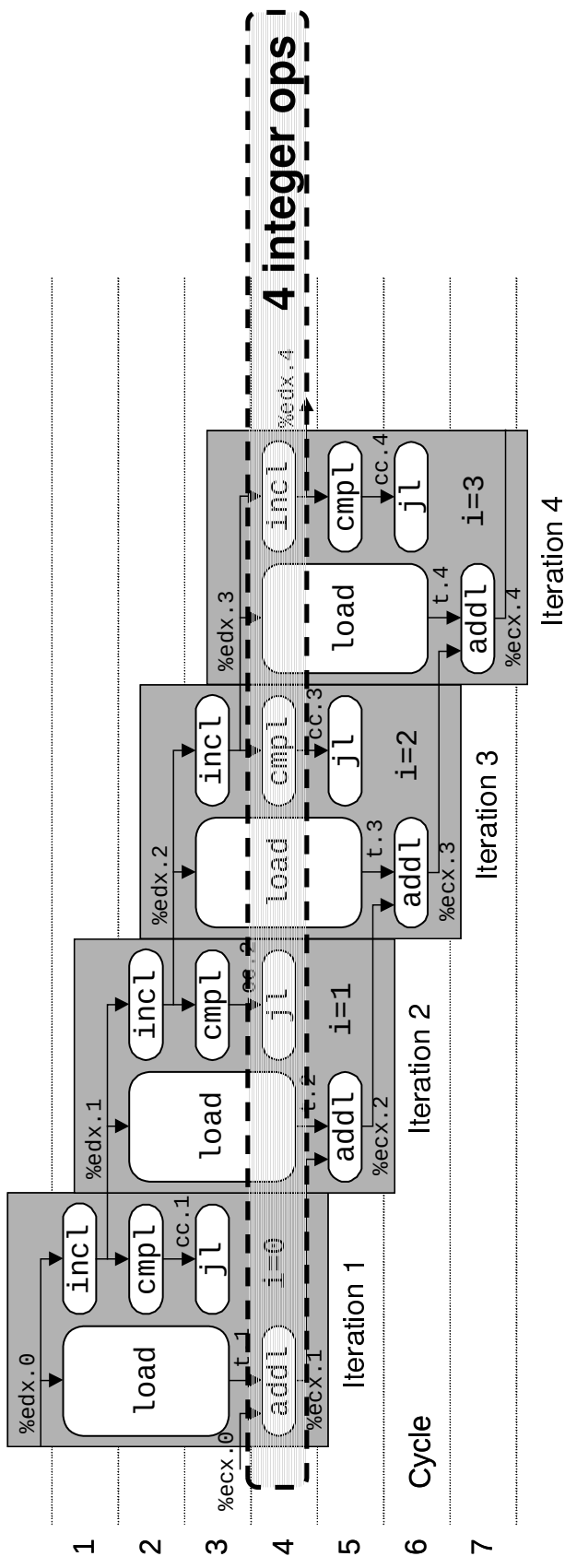
Unlimited Resource Analysis

- Assume operation can start as soon as operands available
- Operations for multiple iterations overlap in time

Performance

- Limiting factor becomes latency of integer multiplier
- Gives CPE of 4.0

4 Iterations of Combining Sum

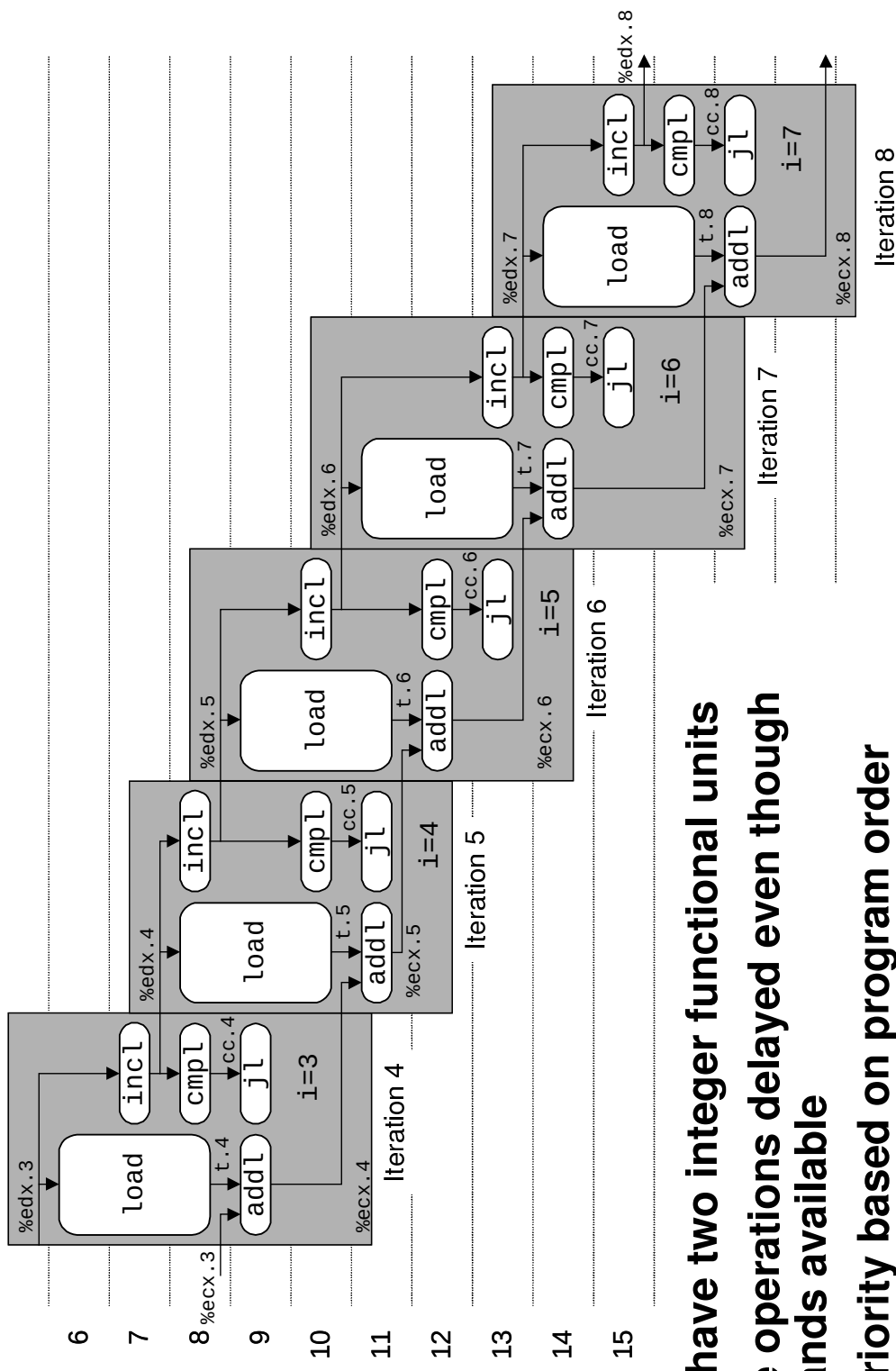


Unlimited Resource Analysis

Performance

- Can begin a new iteration on each clock cycle
- Should give CPE of 1.0
- Would require executing 4 integer operations in parallel

Combining Sum: Resource Constraints



- Only have two integer functional units
- Some operations delayed even though operands available
- Set priority based on program order

Performance

- Sustain CPE of 2.0

Loop Unrolling

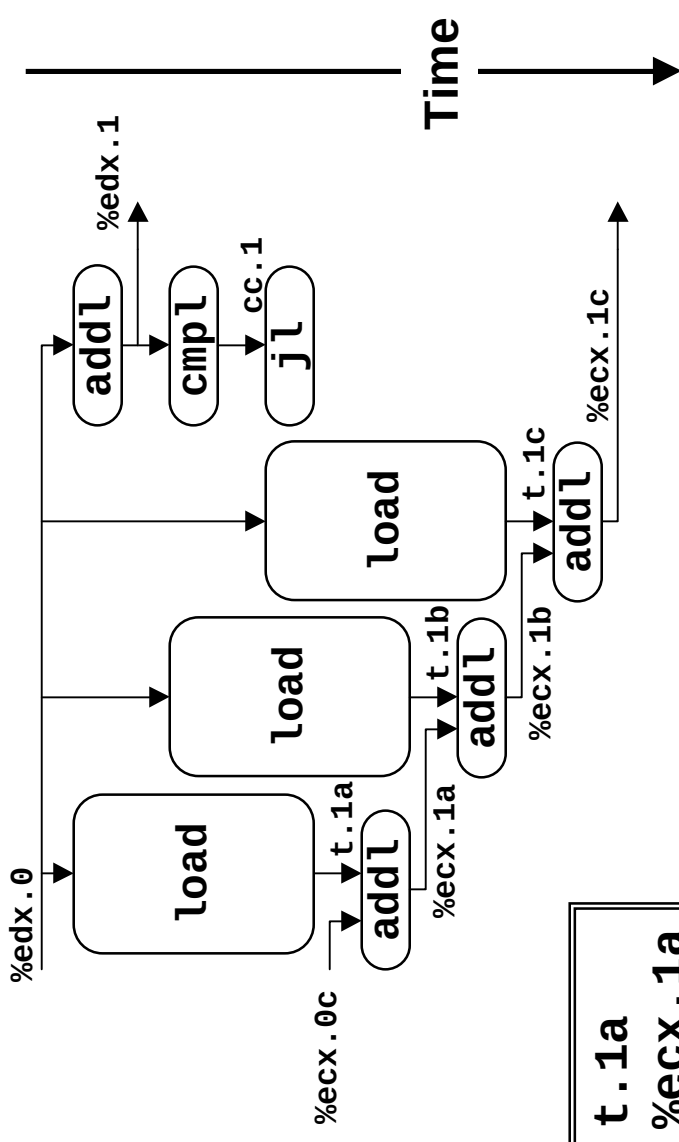
```
void combine5(vec_ptr v, int *dest)
{
    int length = vec_length(v);
    int limit = length-2;
    int *data = get_vec_start(v);
    int sum = 0;
    int i;
    /* Combine 3 elements at a time */
    for (i = 0; i < limit; i+=3) {
        sum += data[i] + data[i+2]
              + data[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        sum += data[i];
    }
    *dest = sum;
}
```

Optimization

- Combine multiple iterations into single loop body
- Amortizes loop overhead across multiple iterations
- Finish extras at end
- Measured CPE = 1.33

Visualizing Unrolled Loop

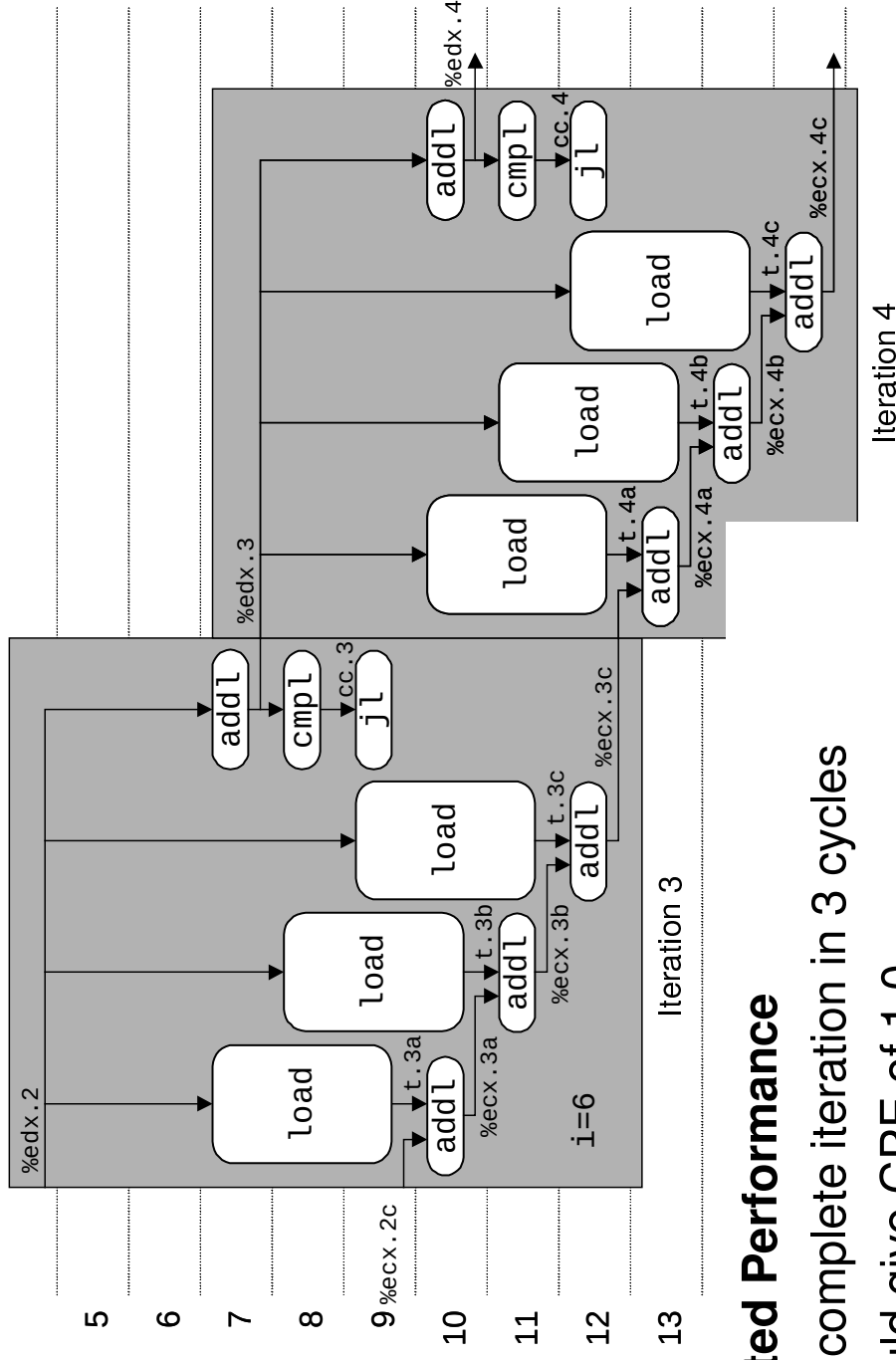
- Loads can pipeline, since don't have dependencies
- Only one set of loop control operations



```

load (%eax,%edx.0,4) → t.1a
iaddl t.1a, %ecx.0c → %ecx.1a
load 4(%eax,%edx.0,4) → t.1b
iaddl t.1b, %ecx.1a → %ecx.1b
load 8(%eax,%edx.0,4) → t.1c
iaddl t.1c, %ecx.1b → %ecx.1c
iaddl $3,%edx.0 → %edx.1
cmpl %esi, %edx.1 → cc.1
jnl-taken cc.1
    
```

Executing with Loop Unrolling



- **Predicted Performance**
 - Can complete iteration in 3 cycles
 - Should give CPE of 1.0
- **Measured Performance**
 - CPE of 1.33
 - One iteration every 4 cycles

Effect of Unrolling

Unrolling Degree		1	2	3	4	8	16
Integer	Sum	2.00	1.50	1.33	1.50	1.25	1.06
Integer	Product	4.00					
FP	Sum	3.00					
FP	Product	5.00					

- **Only helps integer sum for our examples**
 - Other cases constrained by functional unit latencies
- **Effect is nonlinear with degree of unrolling**
 - Many subtle effects determine exact scheduling of operations

Parallel Loop Unrolling

```
void combine6(vec_ptr v, int *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    int *data = get_vec_start(v);
    int x0 = 1;
    int x1 = 1;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x0 *= data[i];
        x1 *= data[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 *= data[i];
    }
    *dest = x0 * x1;
}
```

Code Version

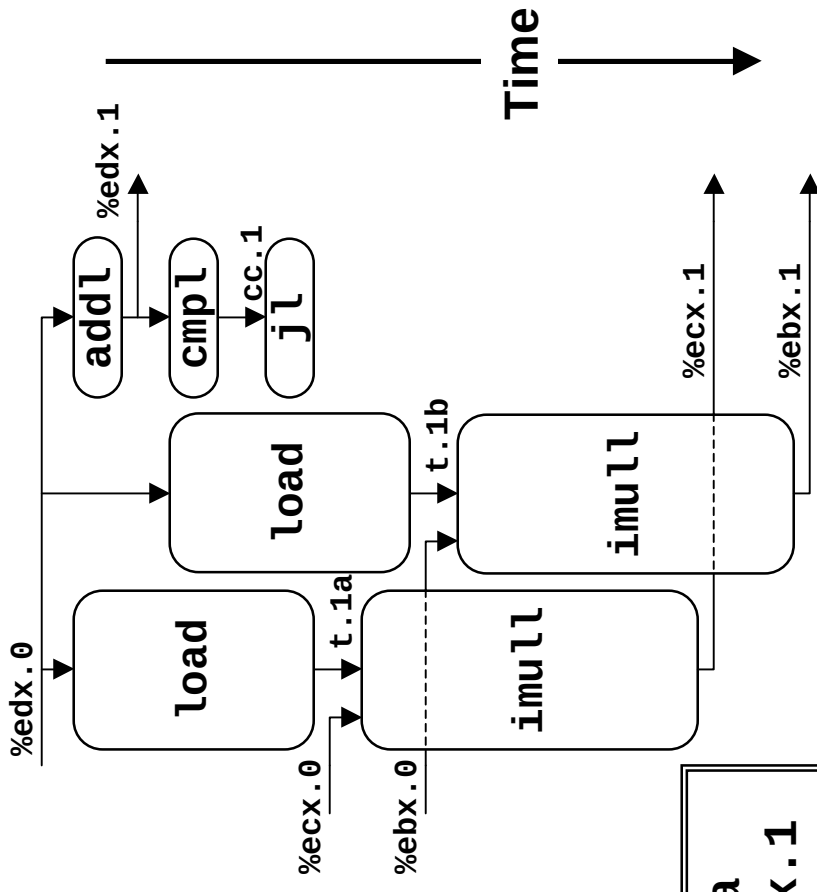
- Integer product

Optimization

- Accumulate in two different sums
 - Can be performed simultaneously
- Combine at end
- Exploits property that integer addition & multiplication are associative & commutative
- FP addition & multiplication not associative, but transformation usually acceptable

Visualizing Parallel Loop

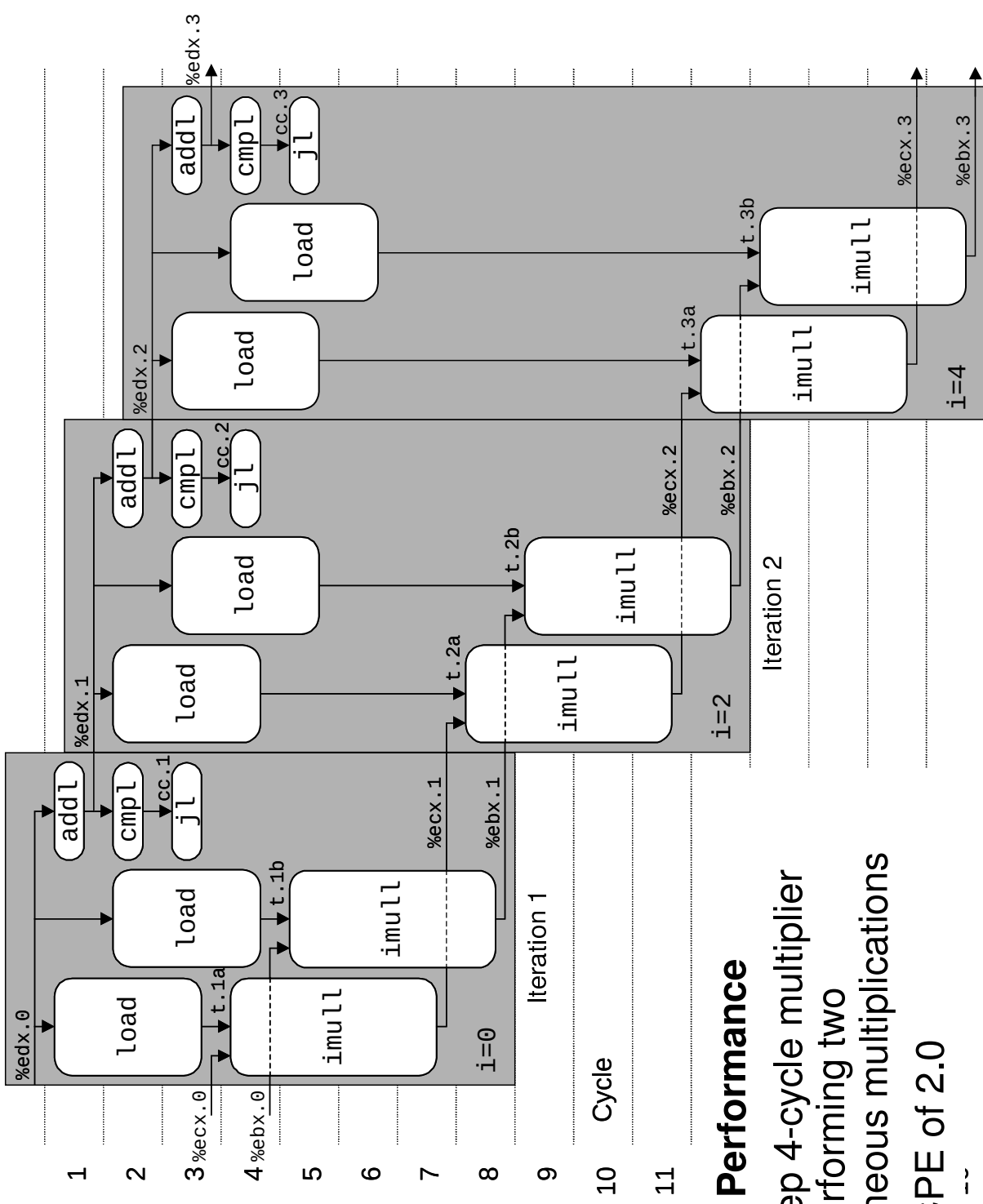
- Two multiplies within loop no longer have data dependency
- Allows them to pipeline



```

load (%eax, %edx.0, 4) → t.1a
imull t.1a, %ecx.0 → %ecx.1
load 4(%eax, %edx.0, 4) → t.1b
imull t.1b, %ebx.0 → %ebx.1
iaddl $2, %edx.0 → %edx.1
cmpl %esi, %edx.1 → cc.1
jl-taken cc.1
    
```

Executing with Parallel Loop



- **Predicted Performance**

- Can keep 4-cycle multiplier busy performing two simultaneous multiplications
- Gives CPE of 2.0

Optimization Results for Combining

Method	Integer		Floating Point	
	+	*	+	*
Abstract -g	42.06	41.86	41.44	160.00
Abstract -O2	31.25	33.25	31.25	143.00
Move vec_length	20.66	21.25	21.15	135.00
data access	6.00	9.00	8.00	117.00
Accum. in temp	2.00	4.00	3.00	5.00
Pointer	3.00	4.00	3.00	5.00
Unroll 4	1.50	4.00	3.00	5.00
Unroll 16	1.06	4.00	3.00	5.00
2 X 2	1.50	2.00	2.00	2.50
4 X 4	1.50	2.00	1.50	2.50
8 X 4	1.25	1.25	1.50	2.00
Theoretical Opt.	1.00	1.00	1.00	2.00
<i>Worst : Best</i>	39.7	33.5	27.6	80.0

Limitations of Parallel Execution

Need Lots of Registers

- To hold sums/products
- Only 6 usable integer registers
 - Also needed for pointers, loop conditions
- 8 FP registers
- When not enough registers, must *spill* temporaries onto stack
 - Wipes out any performance gains
- Not helped by renaming
 - Cannot reference more operands than instruction set allows

Example

- 8 X 8 integer product
- 7 local variables share 1 register

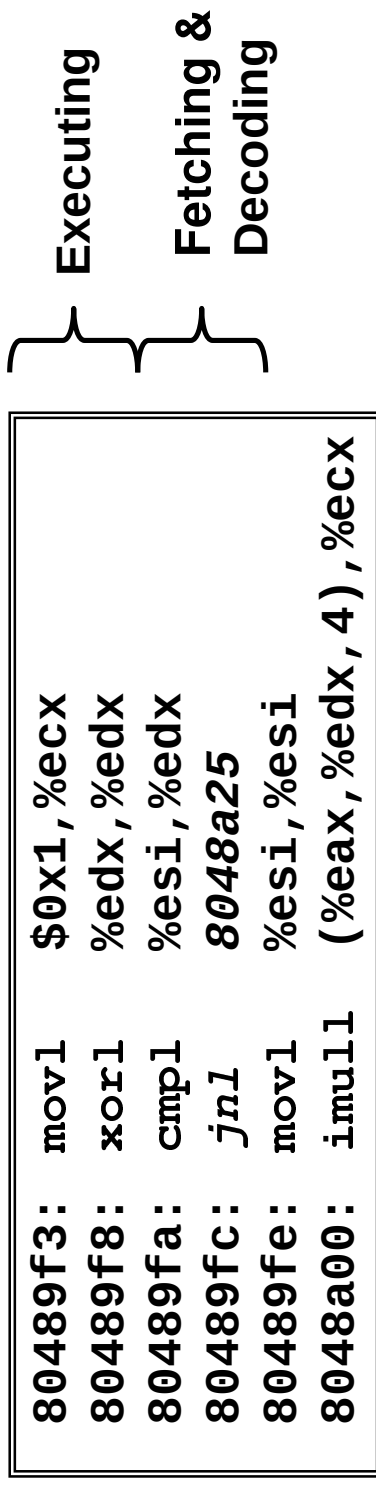
```
.L165:
    imull (%eax), %ecx
    movl -4(%ebp), %edi
    imull 4(%eax), %edi
    movl %edi, -4(%ebp)
    movl -8(%ebp), %edi
    imull 8(%eax), %edi
    movl %edi, -8(%ebp)
    movl -12(%ebp), %edi
    imull 12(%eax), %edi
    movl %edi, -12(%ebp)
    movl -16(%ebp), %edi
    imull 16(%eax), %edi
    movl %edi, -16(%ebp)

    ...
    addl $32, %eax
    addl $8, %edx
    cmpl -32(%ebp), %edx
    jl .L165
```

What About Branches?

Challenge

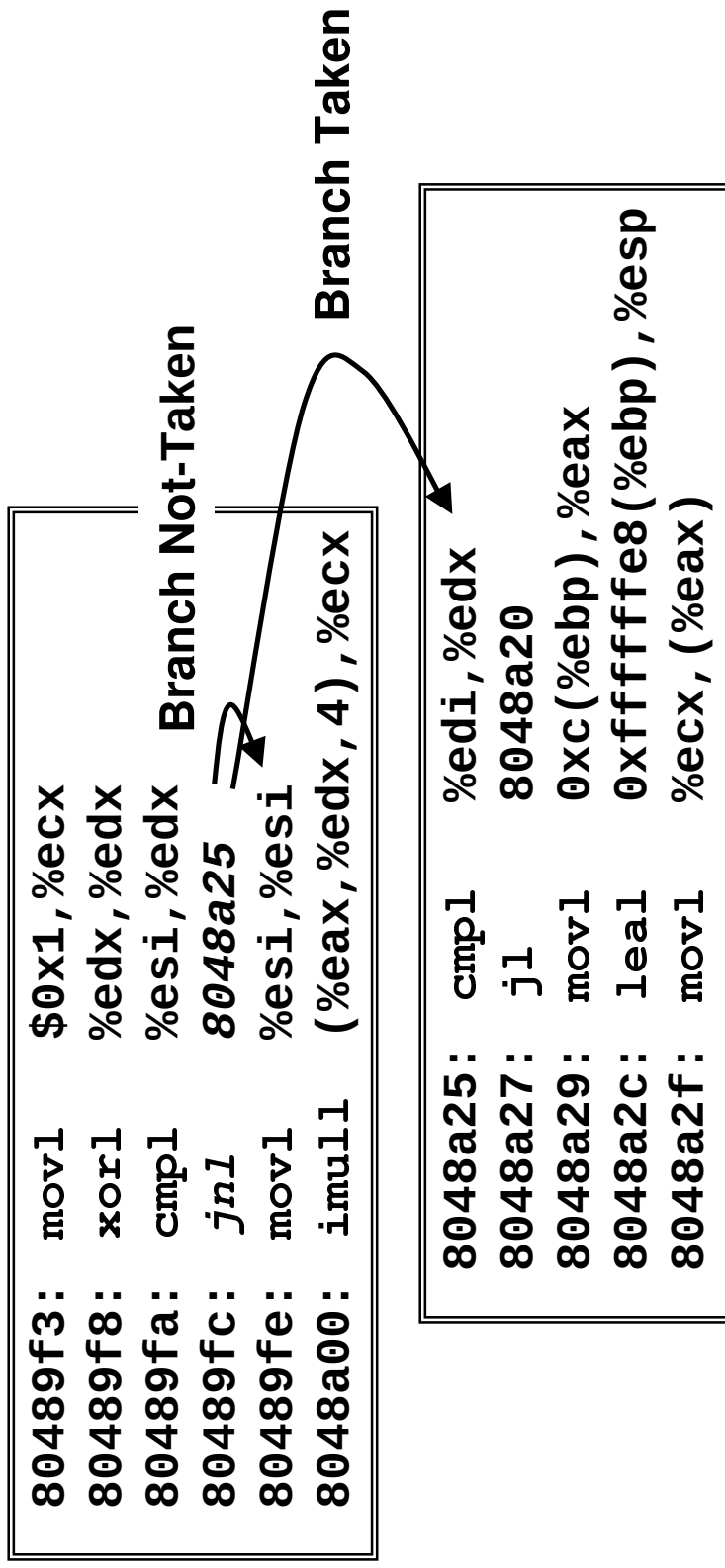
- Instruction Control Unit must work well ahead of Exec. Unit
 - To generate enough operations to keep EU busy



- When encounters conditional branch, cannot reliably determine where to continue fetching

Branch Outcomes

- When encounter conditional branch, cannot determine where to continue fetching
 - Branch Taken: Transfer control to branch target
 - Branch Not-Taken: Continue with next instruction in sequence
- Cannot resolve until outcome determined by branch/integer unit



Branch Prediction

Idea

- Guess which way branch will go
- Begin executing instructions at predicted position
 - But don't actually modify register or memory data

80489f3:	movl	\$0x1,%ecx
80489f8:	xorl	%edx,%edx
80489fa:	cmpl	%esi,%edx
80489fc:	jnl	8048a25
. . .		

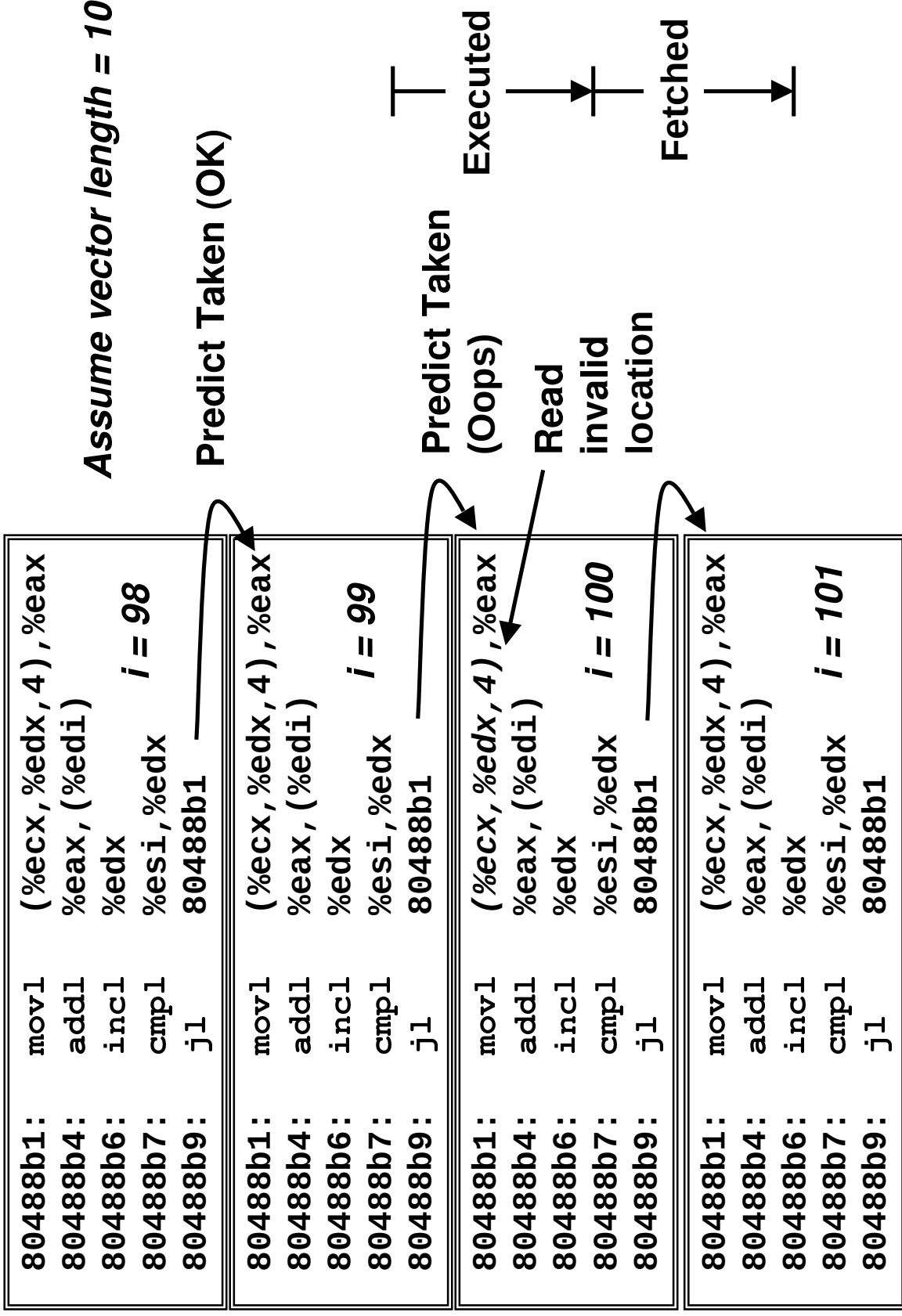
Predict Taken

8048a25:	cmpl	%edi,%edx
8048a27:	jl	8048a20
8048a29:	movl	0xc(%ebp),%eax
8048a2c:	leal	0xffffffe8(%ebp),%esp
8048a2f:	movl	%ecx,(%eax)

Execute

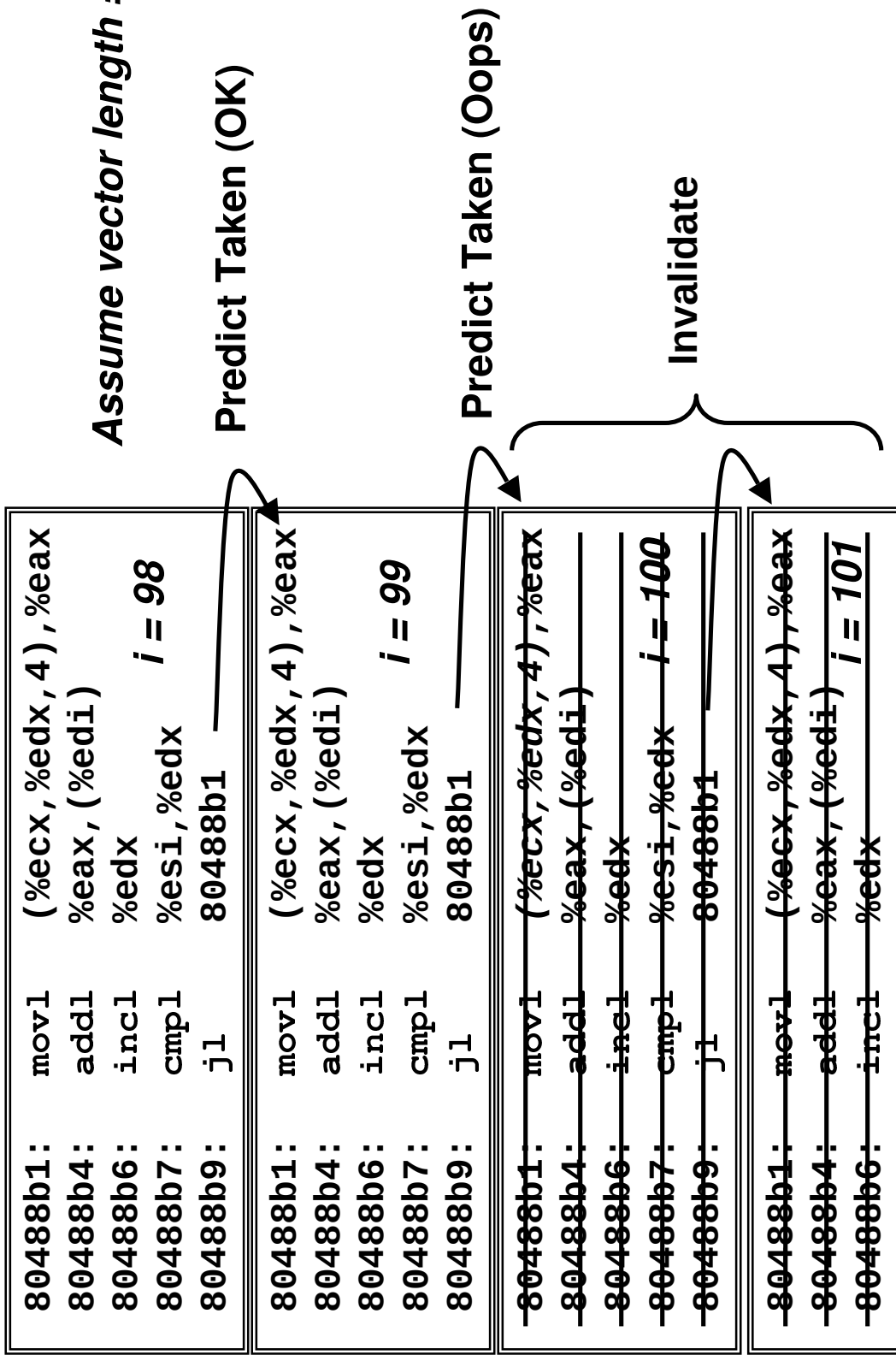
Branch Prediction Through Loop

Assume *vector length* = 100

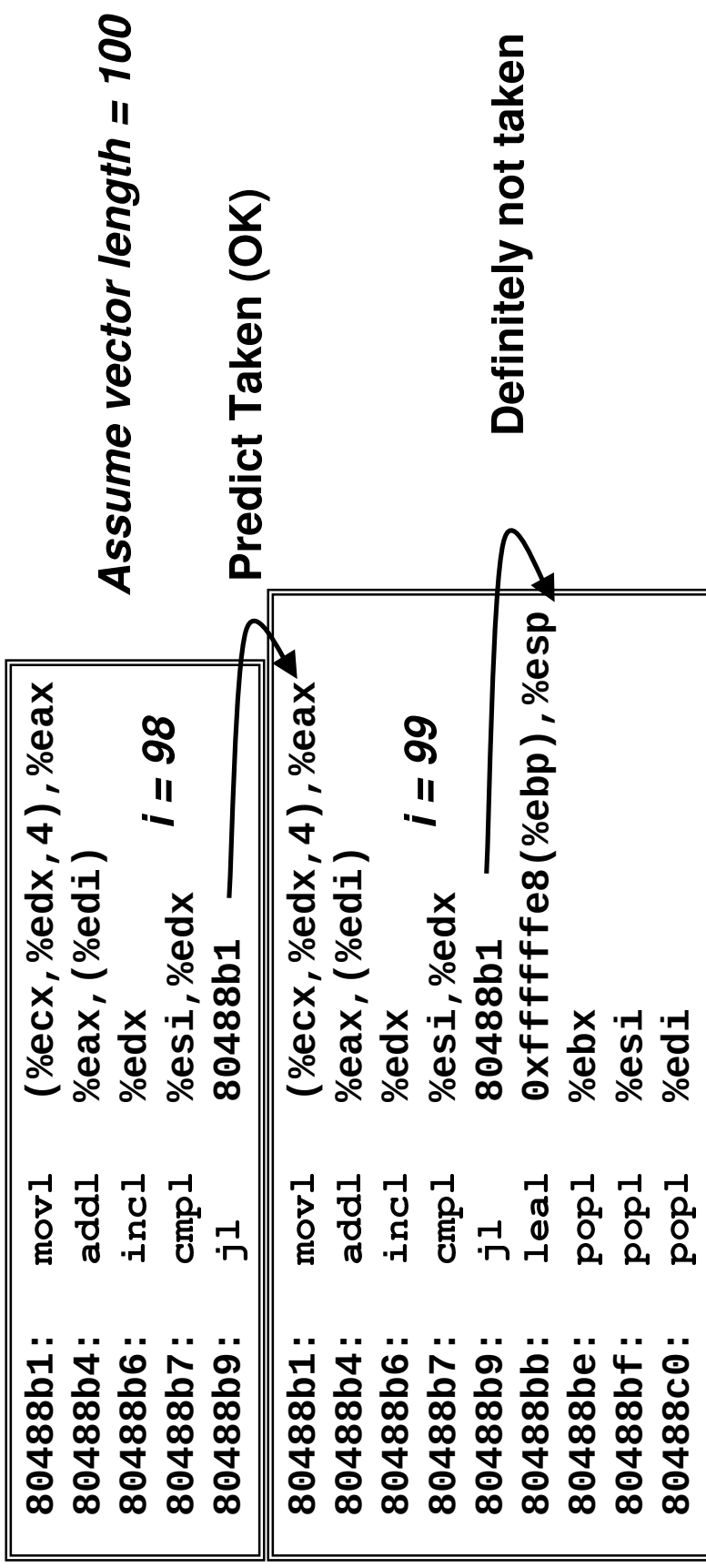


Branch Misprediction Invalidation

Assume vector length = 100



Branch Misprediction Recovery



Performance Cost

- Misprediction on Pentium III wastes ~14 clock cycles
- That's a lot of time on a high performance processor

Results for Alpha Processor

Method	Integer		Floating Point	
	+	*	+	*
Abstract -g	40.14	47.14	52.07	53.71
Abstract -O2	25.08	36.05	37.37	32.02
Move vec_length	19.19	32.18	28.73	32.73
data access	6.26	12.52	13.26	13.01
Accum. in temp	1.76	9.01	8.08	8.01
Unroll 4	1.51	9.01	6.32	6.32
Unroll 16	1.25	9.01	6.33	6.22
4 X 2	1.19	4.69	4.44	4.45
8 X 4	1.15	4.12	2.34	2.01
8 X 8	1.11	4.24	2.36	2.08
Worst : Best	36.2	11.4	22.3	26.7

- Overall trends very similar to those for Pentium III.
- Even though very different architecture and compiler

Machine-Dependent Opt. Summary

Pointer Code

- Look carefully at generated code to see whether helpful

Loop Unrolling

- Some compilers do this automatically
- Generally not as clever as what can achieve by hand

Exposing Instruction-Level Parallelism

- Very machine dependent

Warning:

- Benefits depend heavily on particular machine
- Best if performed by compiler
 - But GCC on IA32/Linux is not very good
- Do only for performance-critical parts of code

Role of Programmer

How should I write my programs, given that I have a good, optimizing compiler?

Don't: Smash Code into Oblivion

- Hard to read, maintain, & assure correctness

Do:

- Select best algorithm
- Write code that's readable & maintainable
 - Procedures, recursion, without built-in constant limits
 - Even though these factors can slow down code
- Eliminate optimization blockers
 - Allows compiler to do its job

Focus on Inner Loops

- Do detailed optimizations where code will be executed repeatedly
- Will get most performance gain here