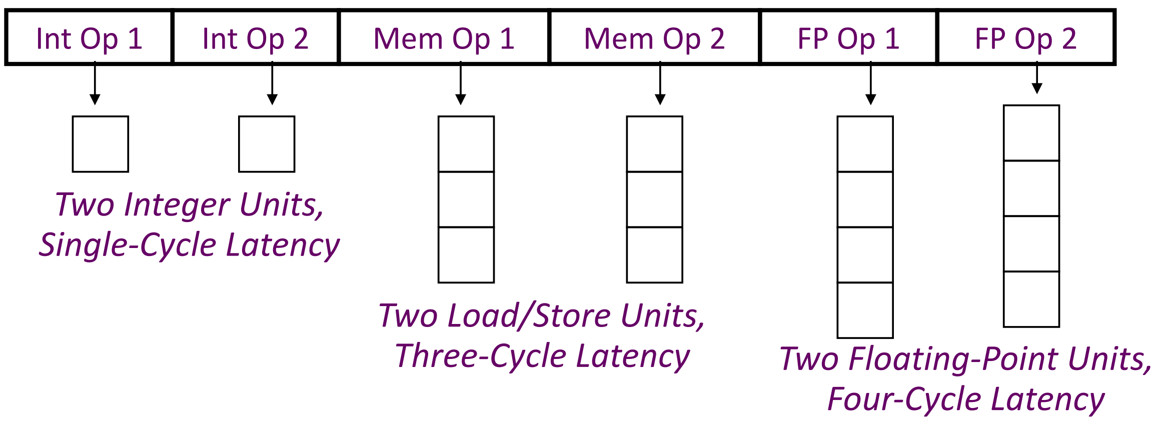
# CSC 631: High-Performance Computer Architecture

Spring 2022

Lecture 7: VLIW

## VLIW: Very Long Instruction Word




- Multiple operations packed into one instruction
- Each operation slot is for a fixed function
- Constant operation latencies are specified
- Architecture requires guarantee of:
  - Parallelism within an instruction => no cross-operation RAW check
  - No data use before data ready => no data interlocks

# VLIW Compiler Responsibilities

- Schedule operations to maximize parallel execution
- Guarantees intra-instruction parallelism
- Schedule to avoid data hazards (no interlocks)
  - Typically separates operations with explicit NOPs



# Loop Execution

```
for (i=0; i<N; i++)
    B[i] = A[i] + C;
```

*Compile*

```
loop:   fld f1, 0(x1)
        add x1, 8
        fadd f2, f0, f1
        fsd f2, 0(x2)
        add x2, 8
        bne x1, x3,
loop
```

*Schedule*

loop:

| Int1 | Int 2 | M1 | M2 | FP+ | FPx |
|---|---|---|---|---|---|
| add x1 | | fld | | | |
| | | | | | |
| | | | | | |
| | | | | fadd | |
| | | | | | |
| | | | | | |
| | | | | | |
| add x2 | bne | fsd | | | |
| | | | | | |

How many FP ops/cycle?

1 fadd / 8 cycles = 0.125

# Loop Unrolling

```
for (i=0; i<N; i++)
   B[i] = A[i] + C;
```

Unroll inner loop to perform 4 iterations at once

```
for (i=0; i<N; i+=4)
{
   B[i]   = A[i] + C;
   B[i+1] = A[i+1] + C;
   B[i+2] = A[i+2] + C;
   B[i+3] = A[i+3] + C;
}
```

Need to handle values of N that are not multiples of unrolling factor with final cleanup loop



# Scheduling Loop Unrolled Code

*Unroll 4 ways*

```
loop:  fld f1, 0(x1)
       fld f2, 8(x1)
       fld f3, 16(x1)
       fld f4, 24(x1)
       add x1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       fsd f5, 0(x2)
       fsd f6, 8(x2)
       fsd f7, 16(x2)
       fsd f8, 24(x2)
       add x2, 32
       bne x1, x3, loop
```

*Schedule* →

| | Int1 | Int 2 | M1 | M2 | FP+ | FPx |
|---|---|---|---|---|---|---|
| loop: | | | fld f1 | | | |
| | | | fld f2 | | | |
| | | | fld f3 | | | |
| | add x1 | | fld f4 | | fadd f5 | |
| | | | | | fadd f6 | |
| | | | | | fadd f7 | |
| | | | | | fadd f8 | |
| | | | fsd f5 | | | |
| | | | fsd f6 | | | |
| | | | fsd f7 | | | |
| | add x2 | bne | fsd f8 | | | |
| | | | | | | |
| | | | | | | |

How many FLOPS/cycle?

4 fadds / 11 cycles = 0.36

# Software Pipelining

*Unroll 4 ways first*

```
loop:  fld f1, 0(x1)
       fld f2, 8(x1)
       fld f3, 16(x1)
       fld f4, 24(x1)
       add x1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       fsd f5, 0(x2)
       fsd f6, 8(x2)
       fsd f7, 16(x2)
       add x2, 32
       fsd f8, -8(x2)
       bne x1, x3, loop
```

| | Int1 | Int 2 | M1 | M2 | FP+ | FPx |
|---|---|---|---|---|---|---|
| prolog | | | fld f1 | | | |
| | | | fld f2 | | | |
| | | | fld f3 | | | |
| | add x1 | | fld f4 | | | |
| | | | fld f1 | | fadd f5 | |
| | | | fld f2 | | fadd f6 | |
| | | | fld f3 | | fadd f7 | |
| | add x1 | | fld f4 | | fadd f8 | |
| iterate, loop: | | | fld f1 | fsd f5 | fadd f5 | |
| | | | fld f2 | fsd f6 | fadd f6 | |
| | | add x2 | fld f3 | fsd f7 | fadd f7 | |
| | add x1 | bne | fld f4 | fsd f8 | fadd f8 | |
| epilog | | | | fsd f5 | fadd f5 | |
| | | | | fsd f6 | fadd f6 | |
| | | add x2 | | fsd f7 | fadd f7 | |
| | | bne | | fsd f8 | fadd f8 | |
| | | | | fsd f5 | | |

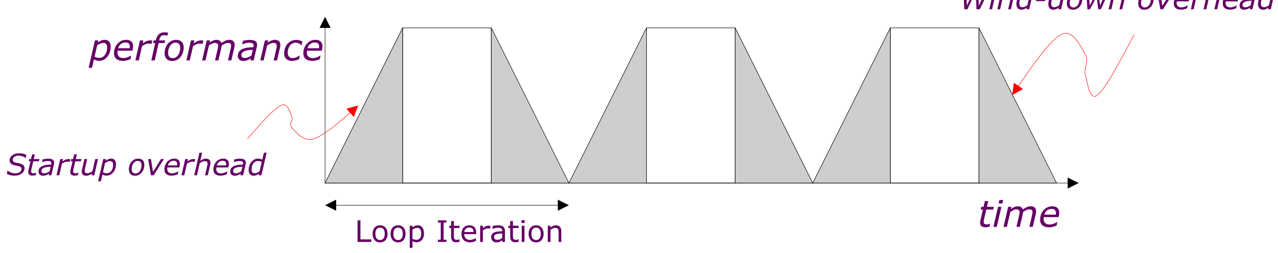
How many FLOPS/cycle?

4 fadds / 4 cycles = 1

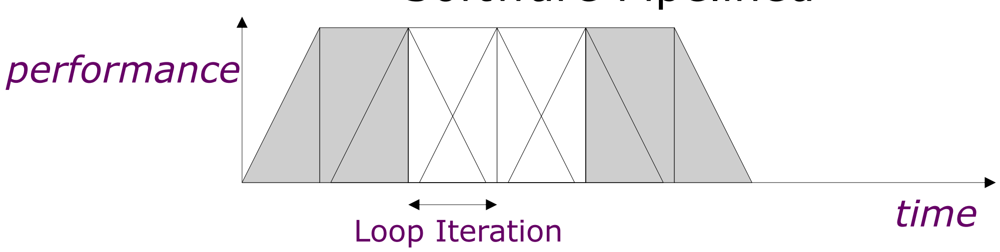


# Software Pipelining vs. Loop Unrolling

## Loop Unrolled

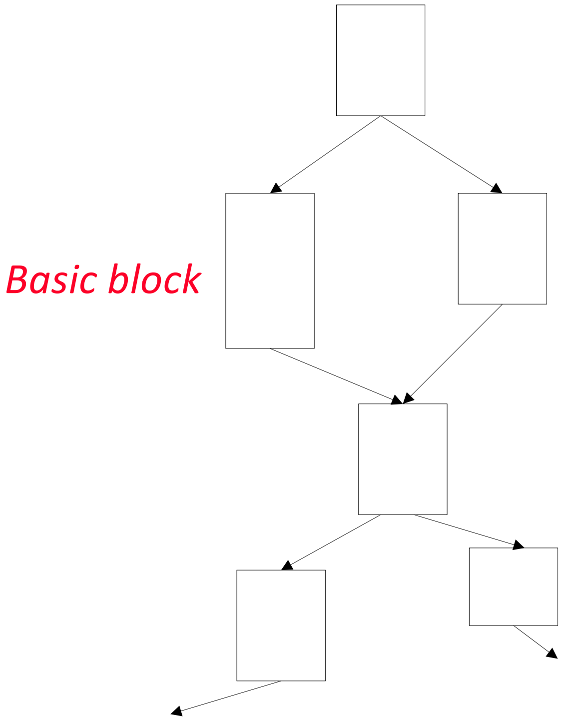



## Software Pipelined




*Software pipelining pays startup/wind-down costs only once per loop, not once per iteration*

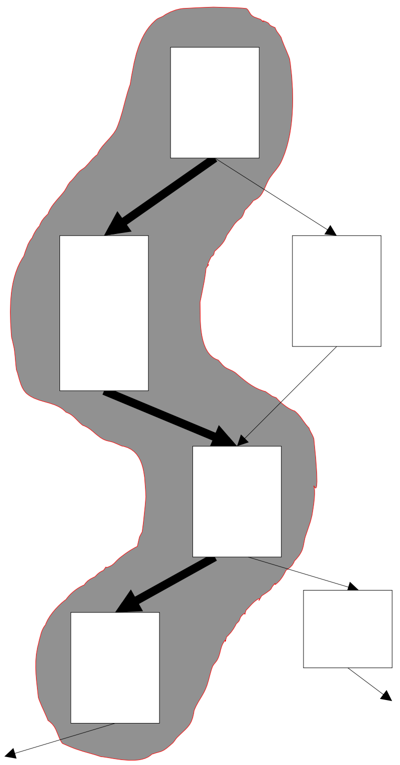
## What if there are no loops?




- Branches limit basic block size in control-flow intensive irregular code
- Difficult to find ILP in individual basic blocks



## Trace Scheduling *[ Fisher,Ellis]*



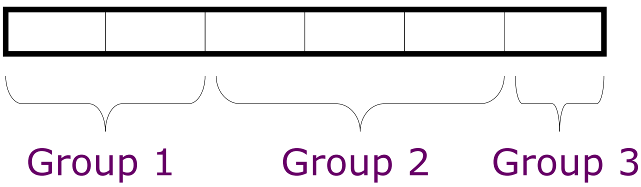
- Pick string of basic blocks, a *trace*, that represents most frequent branch path
- Use profiling feedback or compiler heuristics to find common branch paths
- Schedule whole “trace” at once
- Add fixup code to cope with branches jumping out of trace

# Problems with “Classic” VLIW

- Object-code compatibility
  - have to recompile all code for every machine, even for two machines in same generation
- Object code size
  - instruction padding wastes instruction memory/cache
  - loop unrolling/software pipelining replicates code
- Scheduling variable latency memory operations
  - caches and/or memory bank conflicts impose statically unpredictable variability
- Knowing branch probabilities
  - Profiling requires an significant extra step in build process
- Scheduling for statically unpredictable branches
  - optimal schedule varies with branch path



# VLIW Instruction Encoding



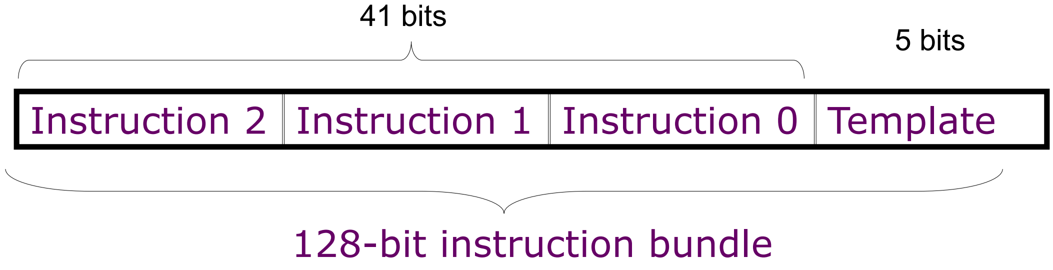

- Schemes to reduce effect of unused fields
  - Compressed format in memory, expand on I-cache refill
    - used in Multiflow Trace
    - introduces instruction addressing challenge
  - Mark parallel groups
    - used in TMS320C6x DSPs, Intel IA-64
  - Provide a single-op VLIW instruction
    - Cydra-5 UniOp instructions

# Intel Itanium, EPIC IA-64

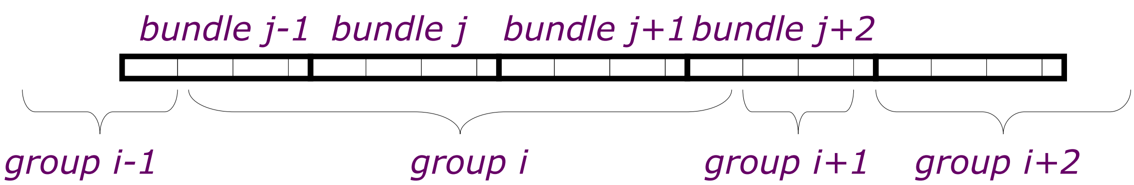
- EPIC is the style of architecture (cf. CISC, RISC)
  - Explicitly Parallel Instruction Computing (really just VLIW)
- IA-64 is Intel's chosen ISA (cf. x86, MIPS)
  - IA-64 = Intel Architecture 64-bit
  - An object-code-compatible VLIW
- Merced was first Itanium implementation (cf. 8086)
  - First customer shipment was expected in 1997 (actually 2001)
  - McKinley, second implementation shipped in 2002
  - Poulson, eight cores, 32nm, 2012
  - Kittson or Itanium 9700, 2017.
    - Similar to Poulson but a higher clock
- Kittson was discontinued in 2019
  - Last ship date July 2021



# IA-64 Instruction Format




- Template bits describe grouping of these instructions with others in adjacent bundles
- Each group contains instructions that can execute in parallel



# Intel Kills Itanium

- Donald Knuth " … *Itanium approach that was supposed to be so terrific—until it turned out that the wished-for compilers were basically impossible to write*."

- "*Intel officially announced the end of life and product discontinuance of the Itanium CPU family on January 30th, 2019*", Wikipedia