EE282
Computer Architecture

Lecture 7: Instruction-Level Parallelism

October 18, 2001

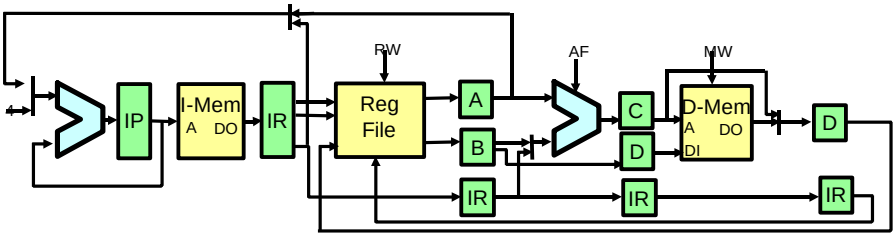
Andrew Wolfe
Computer Systems Laboratory
Stanford University
awolfe@csl.stanford.edu

Today's Lecture

- Instruction-level parallelism
  - covert vs overt parallelism
  - parallelism in hardware and software
  - a question of *scheduling*
  - compilation for ILP
    - code scheduling
    - loop unrolling
- Static approaches to ILP
  - Very-long-instruction-word (VLIW) architectures
    - 2-dimensional schedule
      - when and where
  - Simple, in-order, multiple issue processors

Below 1 Cycle per Instruction

$$T = I \times CPI \times t_{cy} \qquad Performance = \frac{IPC \times frequency}{I}$$



- Existing focus has been to reach 1 IPC
- Never happens for real programs in a basic pipeline

Factors limiting performance

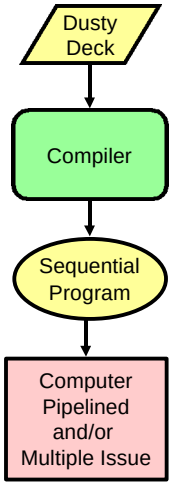
- Data Hazards
- Control Hazards
- Resource Hazards

- In simple 5-stage pipelines - most delays are caused by:
  - Cache misses
  - Branch mispredictions
  - Long instructions (not part of the normal pipeline)

To Unity and Beyond (Below 1 CPI)

- No reason to limit pipelines to process 1 instruction per cycle
- Can predict next several (2-6) instructions and execute them *simultaneously*
- Need to resolve data dependencies
- Several approaches
  - VLIW - compiler schedules instructions
  - Multi-issue - issue instructions in order, but in parallel
  - Superscalar - issue instructions out of order

Parallelism
Overt vs Covert (1 of 2)



Covert approach

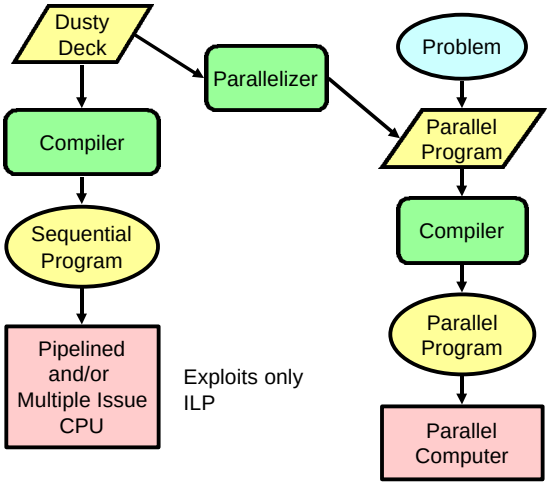
Programmer doesn't see parallelism, just speed

Compiler may discover parallelism (e.g., in loops) and/or reorganize code to remove dependencies

Code generated usually has *sequential semantics* for compatibility with ISA

Hardware executes in parallel while preserving sequential semantics

Parallelism
Overt vs Covert



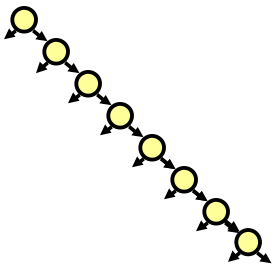
AW/WJD EE282 Lecture 7 10/18/2001 7

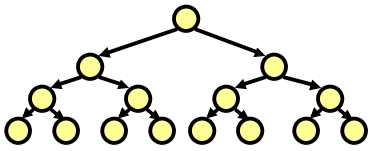
Problem vs Program

Find the maximum element of an array, a

```
max = a[0] ;
for(i=1;i<n;i++) {
  if(a[i] > max) max = a[i] ;
}
return max ;
```

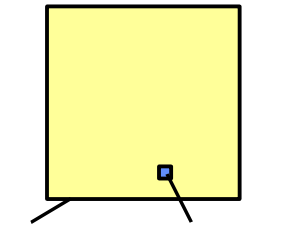
```
for(i=1;i<=logn;i++) {
  m = 1<<(logn-i) ; p = i&1 ; q = p^1 ;
  forall(j=0;j<m;j++){
    a[j,p] = max(a[2*j,q],a[2*j+1,q]);
  }
}
return a[0,p] ;
```

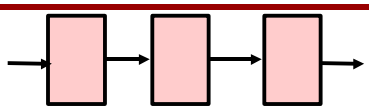


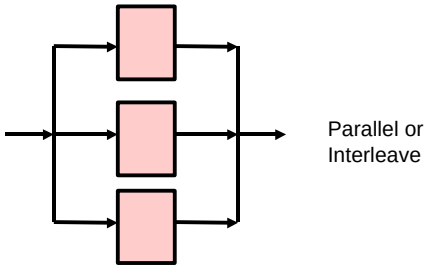


AW/WJD EE282 Lecture 7 10/18/2001 8

Parallelism and Hardware







Technology gives us *lots* of function units

They get only slightly faster each year

The wires get slower

Pipeline or replicate at bit, word, vector, subroutine levels

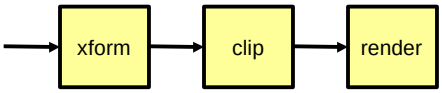
AW/WJD EE282 Lecture 7 10/18/2001 9

Parallelism and Software

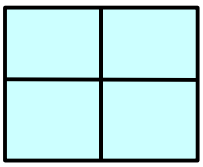
Independent Operations (ILP)

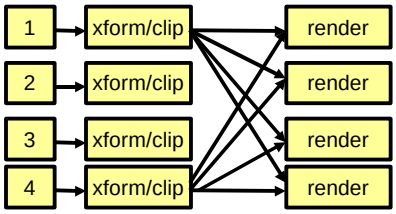
```
a = (b + c) * (d + e + f) ;
```

Function decomposition



Domain decomposition





AW/WJD EE282 Lecture 7 10/18/2001 10

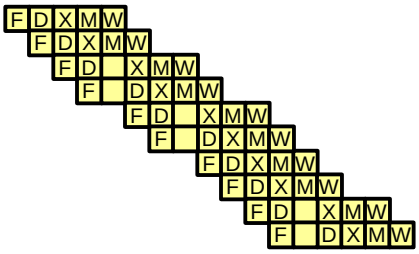
Compilers for Covert Parallelism

```
a = b + c * d ;
e = f + g ;
```

Naive Code

```
LW    R1, C
LW    R2, D
MUL   R3, R1, R2
LW    R4, B
ADD   R5, R4, R3
SW    R5, A
LW    R6, F
LW    R7, G
ADD   R8, R6, R7
SW    R8, E
```

10 instructions
13 cycles
3 load stalls
would be worse if cache miss or if MUL or ADD had latency greater than 1



Compilers for Covert Parallelism (2)

```
a = b + c * d ;
e = f + g ;
```

Rescheduled Code

```
LW    R1, C
LW    R2, D
LW    R4, B
LW    R6, F
MUL   R3, R1, R2
LW    R7, G
ADD   R5, R4, R3
ADD   R8, R6, R7
SW    R5, A
SW    R8, E
```

10 instructions
10 cycles
no load stalls
can tolerate 2-4 cycles of miss latency
can tolerate 2 cycle ADD and MUL

Move loads to the top
Space ALU operations based on pipeline latency

Scheduling can be done by the compiler or at run-time by the hardware

Compilers for Covert Parallelism (3)

```
a = b + c * d ;
e = f + g ;
```

10 operations (4-10 instructions depending on ISA
5 cycles
1 load stall

Parallelized Code

```
LW   R1, C          LW  R2, D          LW  R4, B
LW   R6, F          MUL R3, R1, R2     LW  R7, G
ADD  R5, R4, R3     ADD R8, R6, R7
SW   R5, A          SW  R8, E
```

AW/WJD EE282 Lecture 7 10/18/2001 13

Compilers for Covert Parallelism (4)

```
max = a[0] ;
for(i=1;i<n;i++) {
  if(a[i] > max) max = a[i] ;
}
return max ;
```

```
LOOP:  LW   R1, R2 ;      // a[i]
       SGT  R3, R1, R8 ;  // a[i] > max
       BEQZ R3, NOMAX
       ADDI R8, R1, #0 ;  // update max
NOMAX: ADDI R2, R2, #4 ;  // update a[i] ptr
       ADDI R4, R4, #1 ;  // update i
       SLT  R5, R4, R9 ;  // i < n
       BNEZ R5, LOOP
```

Can only reschedule code inside a *basic block.*

Small basic blocks limit opportunities for scheduling

AW/WJD EE282 Lecture 7 10/18/2001 14

Compilers for Covert Parallelism (5)

```
max = a[0] ;
for(i=1;i<n;i++) {
  if(a[i] > max) max = a[i] ;
}
return max ;
```

```
LOOP:  LW       R1, R2 ;  // a[i]
       SGT      R3, R1, R8 ;       // a[i] > max
IF(R3) ADDI     R8, R1, #0 ;       // update max
       ADDI     R2, R2, #4 ;       // update a[i] ptr
       ADDI     R4, R4, #1 ;       // update i
       SLT      R5, R4, R9 ;       // i < n
       BNEZ     R5, LOOP
```

Predicate conditional to make this all one basic block

```
LOOP:  LW       R1, R2 ;  // a[i]
       ADDI     R4, R4, #1 ;       // update i
       ADDI     R2, R2, #4 ;       // update a[i] ptr
       SGT      R3, R1, R8 ;       // a[i] > max
       SLT      R5, R4, R9 ;       // i < n
IF(R3) ADDI     R8, R1, #0 ;       // update max
       BNEZ     R5, LOOP
```

Reschedule to eliminate stalls and provide slack

Compilers for Covert Parallelism (6)

```
max = a[0] ;
for(i=1;i<n;i++) {
  if(a[i] > max) max = a[i] ;
}
return max ;
```

```
LOOP:   LW      R1, R2 ;       // a[i]
        LW      R11, 4(R2) ;   // a[i+1]
        SGT     R3, R1, R8 ;   // a[i] > max
        ADDI    R2, R2, #8 ;   // update a[i] ptr
IF(R3)  ADDI    R8, R1, #0 ;   // update max
        SGT     R12, R11, R8 ; // a[i+1] > max
IF(R12) ADDI    R8, R11, #0 ;  // update max
        ADDI    R4, R4, #2 ;   // update i
        SLT     R5, R4, R9 ;   // i < n-1
        BNEZ    R5, LOOP
```

Unroll the loop to make an even bigger *basic block*

More opportunities for parallelism

Lower loop overhead 6+4 vs 2(3+4)

This loop has a loop-carried dependence through *max*, more parallelism if loop is not serial.

Compilers for Covert Parallelism (7)

```
for(i=0;i<n;i++) {
  d[i] = a[i]*b[i] + c ;
}
```

No loop-carried dependencies

Can convert data (loop) parallelism into ILP

```
LOOP:   LD      F0, R1 ;        // a[i]
        LD      F2, R2 ;        // b[i]
        ADDI    R1, #8 ;
        ADDI    R2, #8 ;
        MULTD   F4, F0, F2 ;    // a[i] * b[i]
        ADDD    F6, F4, F8 ;    // + c
        SD      F8, R3 ;        // d[i]
        ADDI    R3, #8 ;
        ADDI    R4, #1 ;        // increment i
        SLT     R5, R4, R6 ;    // i<n
        BNEZ    R5, LOOP ;
```

Note that this is converting *easy* parallelism into *hard* parallelism

Unrolled Loop

```
LOOP:   LD      F0, R1 ;        // a[i]
        LD      F2, R2 ;        // b[i]
        LD      F4, 8(R1) ;     // a[i+1]
        LD      F6, 8(R2) ;     // b[i+1]
        ADDI    R1, #16 ;
        ADDI    R2, #16 ;
        MULTD   F8, F0, F2 ;    // a[i] * b[i]
        MULTD   F10, F4, F6 ;   // a[i+1] * b[i+1]
        ADDD    F12, F8, F16 ; // + c
        ADDD    F14, F10, F16 ;    // + c
        SD      F12, R3 ;       // d[i]
        SD      F14, 8(R3) ;    // d[i]
        ADDI    R3, #16 ;
        ADDI    R4, #2 ;        // increment i
        SLT     R5, R4, R6 ;    // i<n-1
        BNEZ    R5, LOOP ;
```

Reduced overhead 10+6 vs 2(5+6)

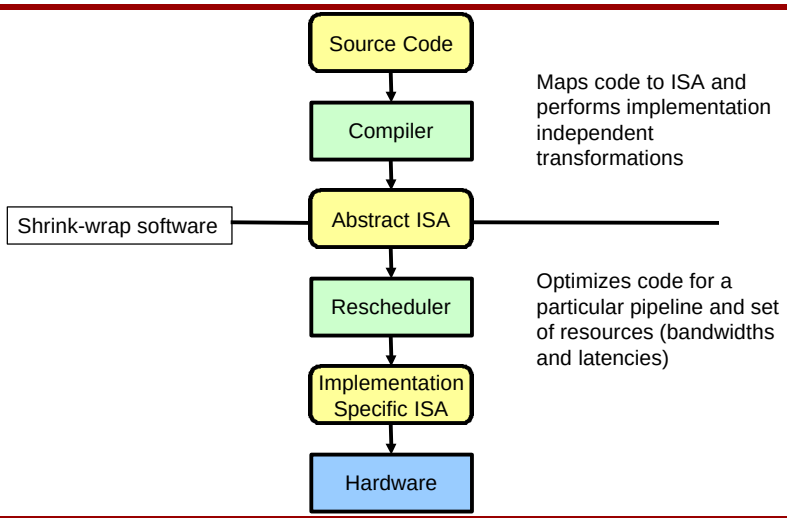
More parallelism and slack

Can unroll further

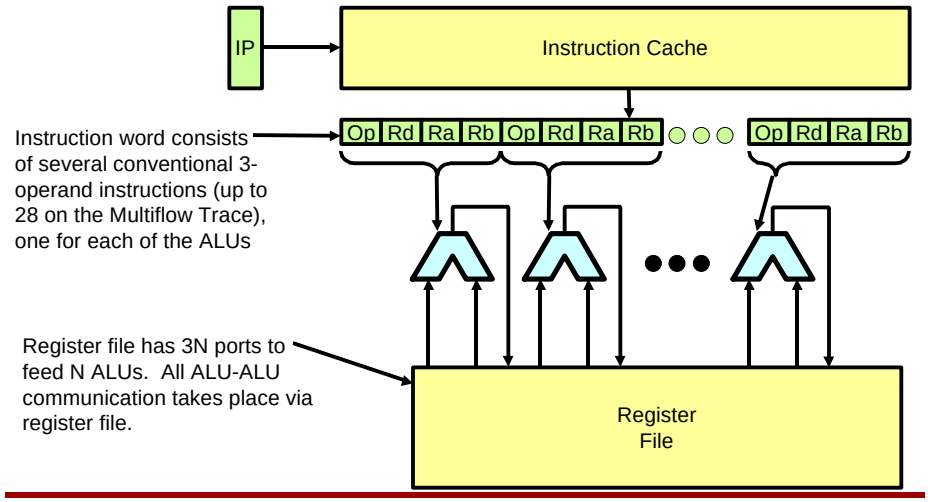
Compilation and ISA

- Efficient compilation requires knowledge of the pipeline structure
  - latency and bandwidth of each operation type
- But a good ISA transcends several implementations with different pipelines
  - should things like a *delayed branch* be in an ISA?
  - should a compiler use the properties of one implementation when compiling for an ISA?
  - do we need a new interface?

An Alternative



Very-Long Instruction Word (VLIW) Computers



AW/WJD EE282 Lecture 7 10/18/2001 21

A Two-Dimensional Schedule

```
for(i=0;i<n;i++) {
  d[i] = a[i]*b[i] + c ;
}
```

Two iterations unrolled

Cycle	MEM1	MEM2	MEM3	MEM4	ADD1	ADD2	ADD3	ADD4	MULT1	MULT2	MULT3	MULT4	BR
1	R1=a[i]	R2=b[i]	R3=a[i+1]	R4=b[i+1]	R10=R10+1	R11=R11+16	R12=R12+16						
2									R5=R1*R2	R6=R3*R4			
3					R7=R5+R9	R8=R6+R9	R14=R10<R15						
4	d[i]=R7	d[i+1]=R8				R13=R13+16							BNEZ 1

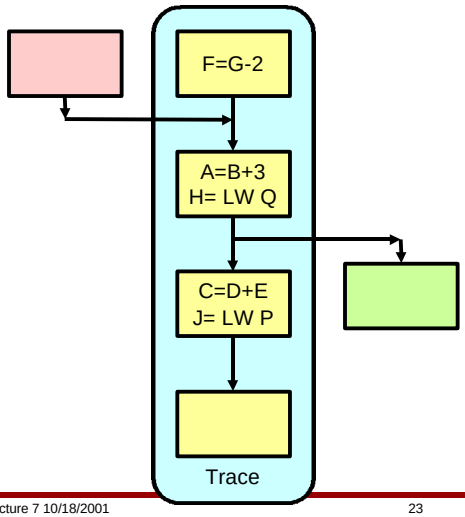
16 operations in 4 cycles
Average ILP = 4
Mostly NOPs, occupancy 16/52 = 31%

With max unrolling, limited by ADD to 1.75 cycles (70%)

AW/WJD EE282 Lecture 7 10/18/2001 22

Trace Scheduling

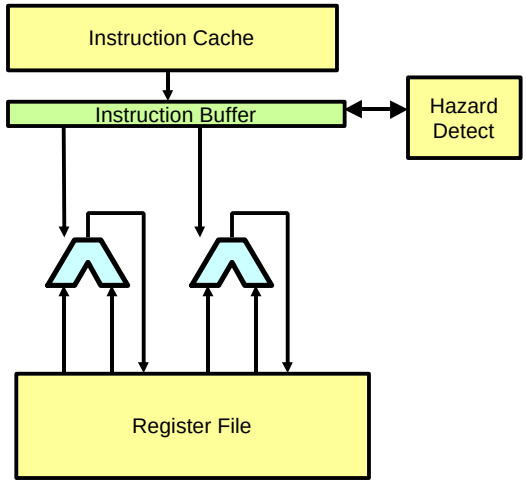
- Easier to find ILP in a longer sequence
  - more operations to choose from
  - more parallel expressions
- Most programs have basic blocks
- Fuse several sequences together along a *trace*
- Allow code motion past basic block boundaries
  - Fixup code when entering or exiting trace
  - speculative loads above branch



VLIW Pros and Cons

- Pros
  - Very simple hardware
    - no dependency detection
    - simple issue logic
    - just ALUs and register file
  - Potentially exploits large amounts of ILP
- Cons
  - Lockstep execution (static schedule)
    - very sensitive to long latency operations (cache misses)
  - Global register file hard to build
  - Lots of NOPs
    - poor code 'density'
    - I-cache capacity and bandwidth compromised
  - Must recompile sources
  - Implementation visible through ISA

Multiple Issue



Multiple Issue (Details)

- Superficially looks like VLIW but:
  - Dependencies and structural hazards checked at run-time
  - Can run existing binaries
    - must recompile for performance, not correctness
- More complex issue logic
  - Swizzle next N instructions into position
  - Check dependencies and resource needs
  - Issue M <= N instructions that can execute in parallel

Example Multiple Issue

Issue rules: at most 1 LD/ST, at most 1 floating op

Latency: LD - 3, int-1, F*-6, F+-3

```
LOOP:   LD      F0, R1 ;        // a[i]
        LD      F2, R2 ;        // b[i]
        LD      F4, 8(R1) ;     // a[i+1]
        LD      F6, 8(R2) ;     // b[i+1]
        ADDI    R1, #16 ;
        ADDI    R2, #16 ;
        MULTD   F8, F0, F2 ;    // a[i] * b[i]
        MULTD   F10, F4, F6 ;   // a[i+1] * b[i+1]
        ADDD    F12, F8, F16 ; // + c
        ADDD    F14, F10, F16 ;    // + c
        SD      F12, R3 ;       // d[i]
        SD      F14, 8(R3) ;    // d[i]
        ADDI    R3, #16 ;
        ADDI    R4, #2 ;        // increment i
        SLT     R5, R4, R6 ;    // i<n-1
        BNEZ    R5, LOOP ;
```


Rescheduled for Multiple Issue

Issue rules: at most 1 LD/ST, at most 1 floating op

Latency: LD - 3, int-1, F*-6, F+-3

```
LOOP:   LD      F0, R1 ;        // a[i]
        ADDI    R1, #16 ;
        LD      F2, R2 ;        // b[i]
        ADDI    R2, #16 ;
        LD      F4, -8(R1) ;    // a[i+1]
        ADDI    R4, #2 ;        // increment i
        LD      F6, -8(R2) ;    // b[i+1]
        MULTD   F8, F0, F2 ;    // a[i] * b[i]
        ADDI    R3, #16 ;
        MULTD   F10, F4, F6 ;   // a[i+1] * b[i+1]
        SLT     R5, R4, R6 ;    // i<n-1
        ADDD    F12, F8, F16 ; // + c
        SD      F12, -16(R3) ; // d[i]
        ADDD    F14, F10, F16 ;    // + c
        SD      F14, -8(R3) ;   // d[i]
        BNEZ    R5, LOOP ;
```

Multiple Issue vs VLIW

- More complex issue logic
  - check dependencies
  - check structural hazards
  - issue variable number of instructions (0-N)
  - shift unissued instructions over
- Able to run existing binaries
  - recompile for performance, not correctness
- Datapaths identical
  - but bypass requires detection
- Neither VLIW or multiple-issue can schedule around run-time variation in instruction latency
  - cache misses
- Dealing with run-time variation requires run-time or *dynamic* scheduling

Next Time

- Dynamic Scheduling
  - Out of order issue
  - Register renaming
    - static single assignment
    - gets rid of WAW and WAR hazards
    - two approaches
  - Reservation stations
    - wait for operands and execution unit
    - distributed *issue* decision
  - Reorder buffer
    - need to *commit* or *retire* instructions in order
  - Examples from the 60s and today