Static Optimizations
(aka: the complier)

Dr. Mark Brehob

EECS 470

Announcements

- Quiz
  - Tuesday 4/2
  - Coverage the same as the last homework
    - Or maybe a bit from earlier

The big picture

- We’ve spent a lot of time learning about dynamic optimizations
  - Finding ways to improve ILP in hardware
    - Out-of-order execution
    - Branch prediction
- But what can be done statically (at compile time)?
  - As hardware architects it behooves us to understand this.
    - Partly so we are aware what things software is likely to be better at.
    - But partly so we can find ways to find hardware/software “synergy”

Some ways a compiler can help

- Improve locality of data
- Remove instructions that aren't needed
- Reduce number of branches executed
- Many others

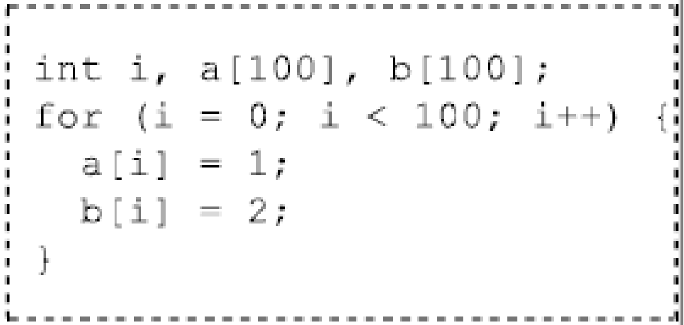
Improve locality of reference

- Examples:
  - **Loop interchange**—flip inner and outer loops

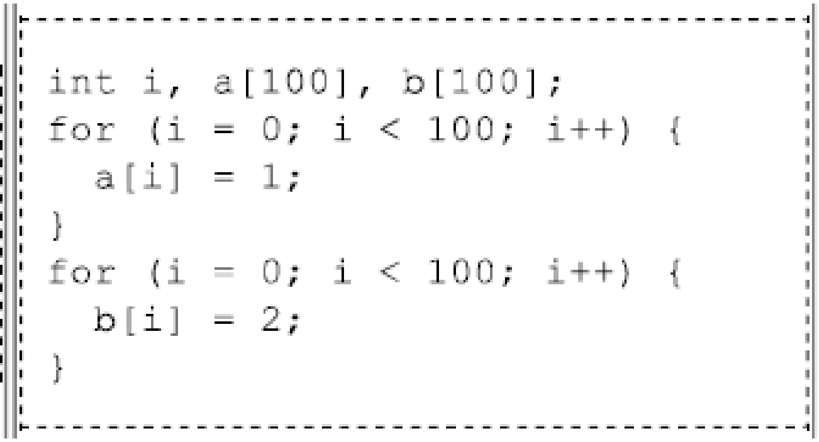
```
for i from 0 to 10
  for j from 0 to 20
     a[j,i] = i + j

for j from 0 to 20
  for i from 0 to 10
    a[j,i] = i + j
```

  - **Loop fission**—split into multiple loops



```
int i, a[100], b[100];
for (i = 0; i < 100; i++) {
  a[i] = 1;
  b[i] = 2;
}
```



```
int i, a[100], b[100];
for (i = 0; i < 100; i++) {
  a[i] = 1;
}
for (i = 0; i < 100; i++) {
  b[i] = 2;
}
```

Some examples taken from Wikipedia

Removing code (1/2)

- **Register optimization**
  - Registers are fast, and doing “spills and fills” is slow.
  - So keep the data likely to be used next in registers.

- **Loop invariant code motion**
  - Move recomputed statements outside of the loop.

```
for (int i=0; i<n; i++) {
        x = y+z;
        a[i] = 6*i+x*x;
}

x = y+z;
for (int i=0; i<n; i++) {
        a[i] = 6*i+x*x;
}
```

Removing code (2/2)

- **Common sub-expression elimination**
  - `(a + b) - (a + b)/4`
    - Just compute a+b once.
- **Constant folding**
  - Replace `(3+5)` with **8**.

Reducing number of branches executed

- **Using predicates or CMOVs instead of short branches**
- **Loop unrolling**

```
for(i=0;i<10000;i++)
{
   A[i]=B[i]+C[i];
}

for(i=0;i<10000;i=i+2)
{
   A[i]=B[i]+C[i];
   A[i+1]=B[i+1]+C[i+1];
}
```

We'll *mostly* focus on one thing

- **"Hoist" loads**
  - That is move the loads up so if there is a miss we can hide that latency.
    - Very similar goal to our OoO processor.

```
xxxxx
xxxxx
LD R1=MEM[x]
R2=R1+R3
```

→

```
LD R1=MEM[x]
xxxxx
xxxxx
R2=R1+R3
```

What limits our ability to hoist a load?

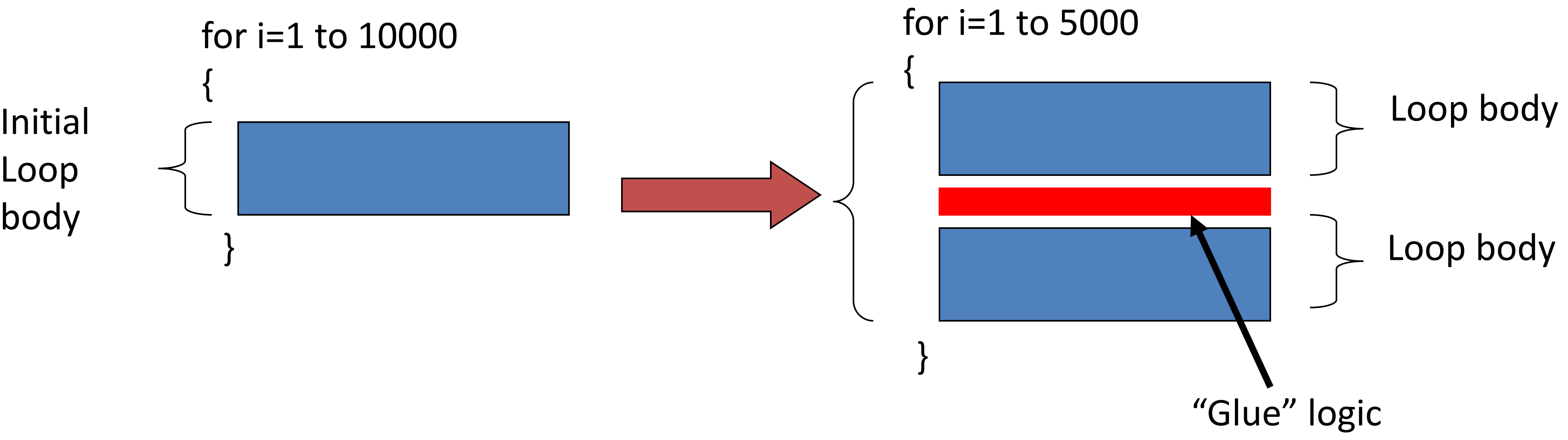
- ______________________________

- ______________________________

- ______________________________

Create room to move code around

- Loop unrolling
  - The idea is to take a loop (usually a short loop) and do two or more iterations in a single loop body.



Unroll this loop

```
for(i=0;i<10000;i++)
   {
     A[i]=B[i]+C[i];
     n+=A[i];
   }
```

Glue logic? Reduce operations?

What does unrolling buy us?

- Reduces number of branches
  - Less to (mis-)predict
  - If not predicting branches (say cheap embedded processor) very helpful!
  - If limited number of branches allowed in ROB at a time, reduces this problem.
- Can schedule for pipeline better
  - If superscalar might be best to combine certain operations. Loop unrolling adds flexibility

What does it cost us?

- Code space.
  - Mainly worried about impact on I-cache hit rate. But L2 or DRAM impact if unroll too much!
- If loop body has branches in it can hurt branch prediction performance.
- Other?

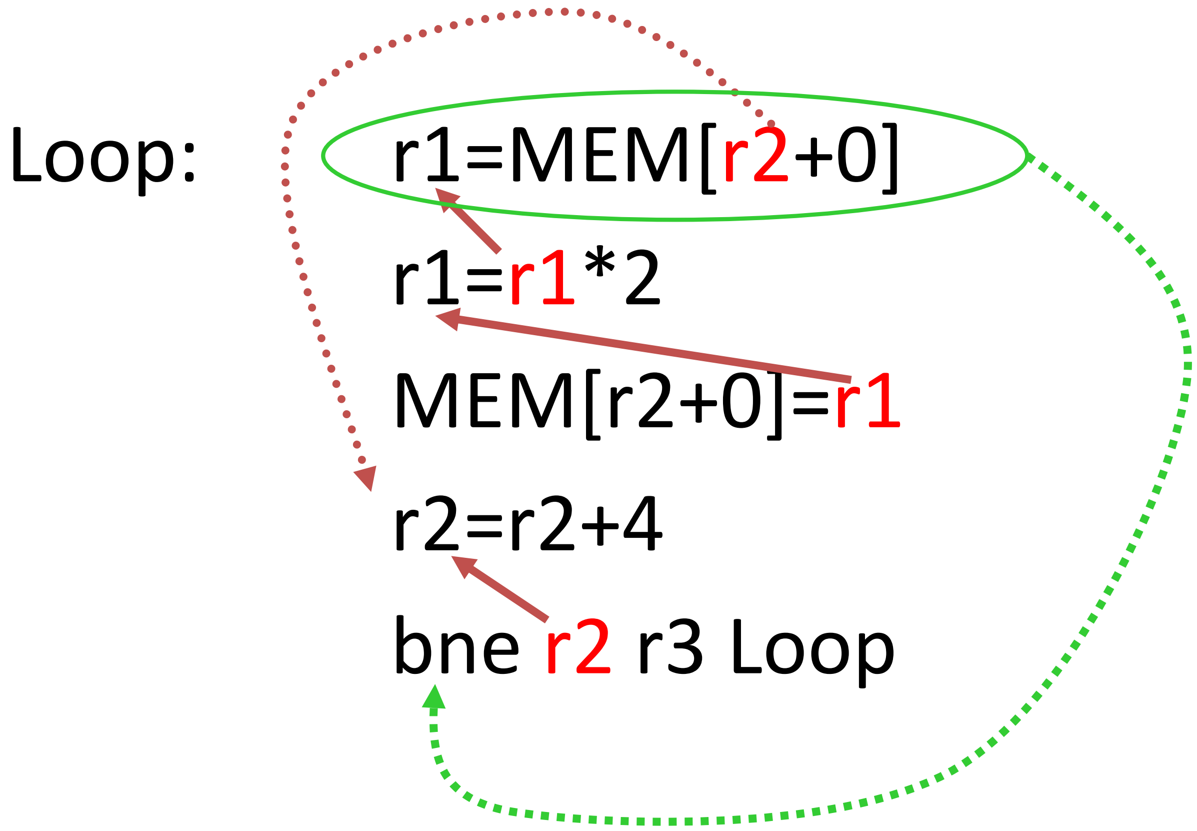
Another one to unroll.

```
for(i=0;i<99999;i++)
   {
     A[i]=B[i]+C[i];
     B[i+1]=C[i]+D[i];
   }
```

One more to do

```
while (B[i]!=0)
   {
    i++;
    A[i]=B[i]+C[i];
    B[i+1]=C[i]+D[i];
   }
```

How about this code?



```
Loop:   r1=MEM[r2+0]
        r1=r1*2
        MEM[r2+0]=r1
        r2=r2+4
        bne r2 r3 Loop
```

We'll come back to this later…

Other ILP techniques

- Consider an in-order superscalar processor executing the following code:

```
R1=16        //A
R2=R1+5      //B
R3=14        //C
R4=R3+5      //D
```

  - Without OoO we would execute A, BC, D.
  - Note that A&B are independent of C&D. So ordering ACBD would let us do AC, BD.
- Thus, the simple action of reordering instructions can increase ILP.

So…

- We can expose ILP by
  - Unrolling loops
  - Reordering code
    - To increase # of independent instructions near each other
    - To move a load (or other high-latency instruction) from its use.
  - What limits reordering options?

The limits of hoisting loads (again)

- Moving code outside of its “basic block” is scary
    - In other words, moving code past branches or branch targets can give wrong execution
        - Loads or stores might go to invalid locations
        - Need to be sure don’t trash a needed register.
- Also
    - Moving loads past stores is scary
        - What if store wrote to that address
- The problem is that we don’t have the recovery mechanisms we do in hardware
    - After all, the program ***<u>specifies</u>*** behavior! How do we know when the specified behavior is “wrong”?
        - In hardware it is fairly easy…

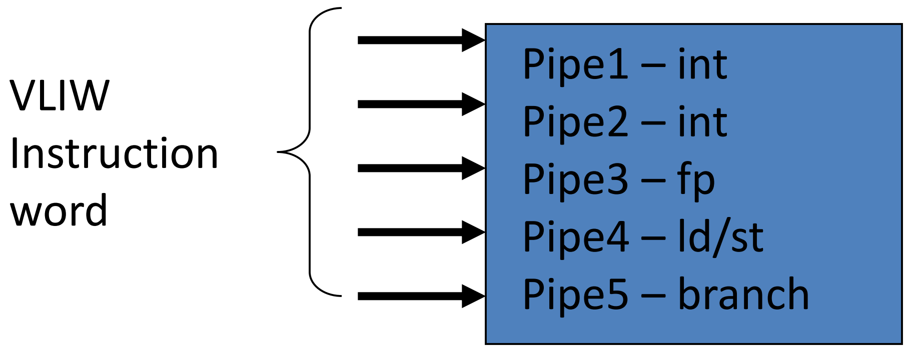
Static dependency checking

- A superscalar processor has to do certain dependency checking at issue (or dispatch)
  - Is a given set of instructions dependent on each other?
  - If ALU resources are shared are there enough resources?

- Many of these issues can be resolved at compile time.
  - What can’t be resolved?
  - Once resolved, how do you tell the CPU?

One static solution: VLIW

- Have a bunch of pipelines, usually with different functional units.
  - Each "instruction" actually contains directions for all the pipelines.
  - (Thus the "very long instruction")



What’s good about VLIW?

- Compiler does all dependency checking, including structural hazards!
  - No dependence checking makes the hardware a lot simpler!
    - Reduces mis-prediction penalty.
    - Saves power
    - May save area!
- Since the compiler can also reorder instructions we may be able to make good use of the pipes.

So what’s bad?

- Code density
  - If you can’t fill a given pipe, need a no-op.
  - To get the ILP needed to be able to fill the pipe, often need to unroll loops.
- When a newer processor comes out, 100% compatibility is hard
  - Word length may need to change
  - Structural dependencies may be different

Conditional execution

- Conditional execution (we've done this before!)

```
        bne r1 r2 skip
        r4=r5+r6
skip:   r7=r4+r12
```

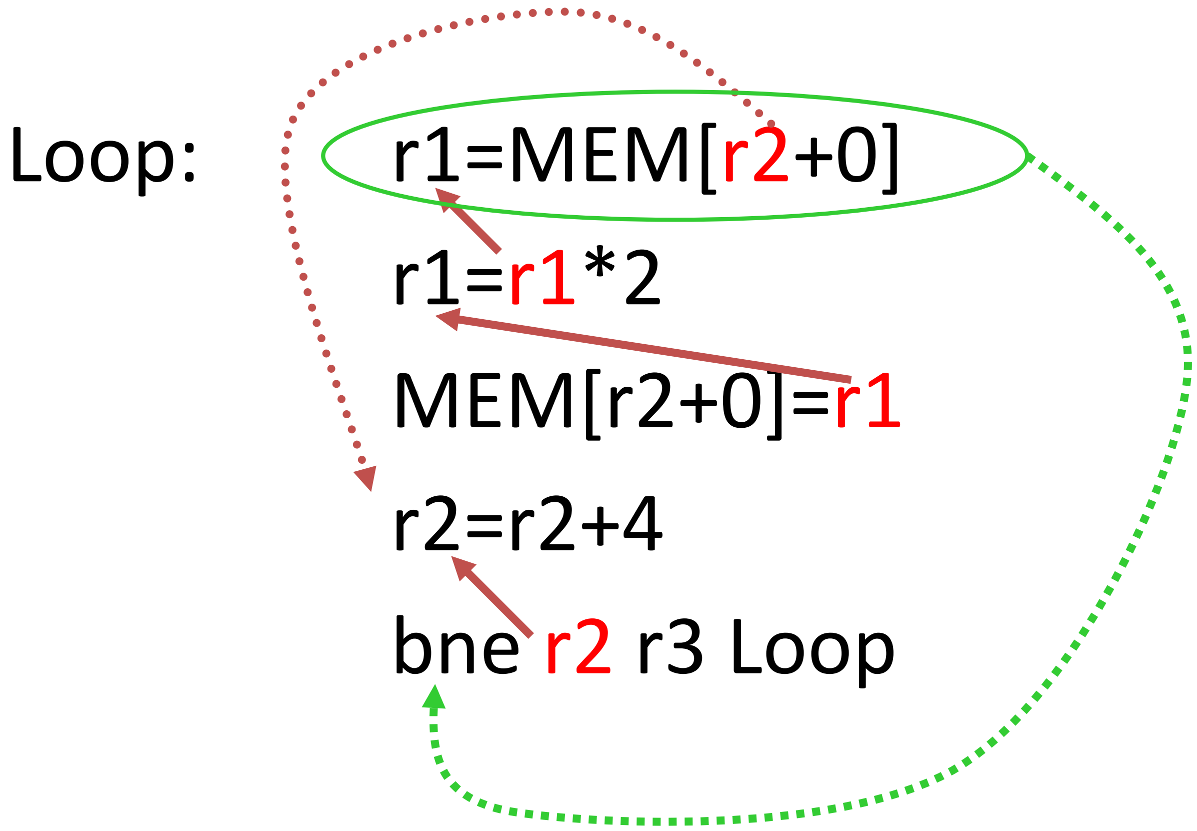
```
r8=cmp(r1,r2)
if(r8) r4=r5 + r6
r7=r4+r12
```

```
-or-
```

```
r8=cmp(r1,r2)
r9=r5+r6
cmov (r8, r4 ← r9)
r7=r4+r12
```

Software pipelining

Code example from before



```
Loop:    r1=MEM[r2+0]
         r1=r1*2
         MEM[r2+0]=r1
         r2=r2+4
         bne r2 r3 Loop
```

And one name dependency on r2.

Also the store is dependent on r2 but hidden…

ILP?

```
r1=MEM[r2+0]      //A
r1=r1*2           //B
MEM[r2+0]=r1      //C
r2=r2+4           //D
bne r2 r3 Loop    //E
```

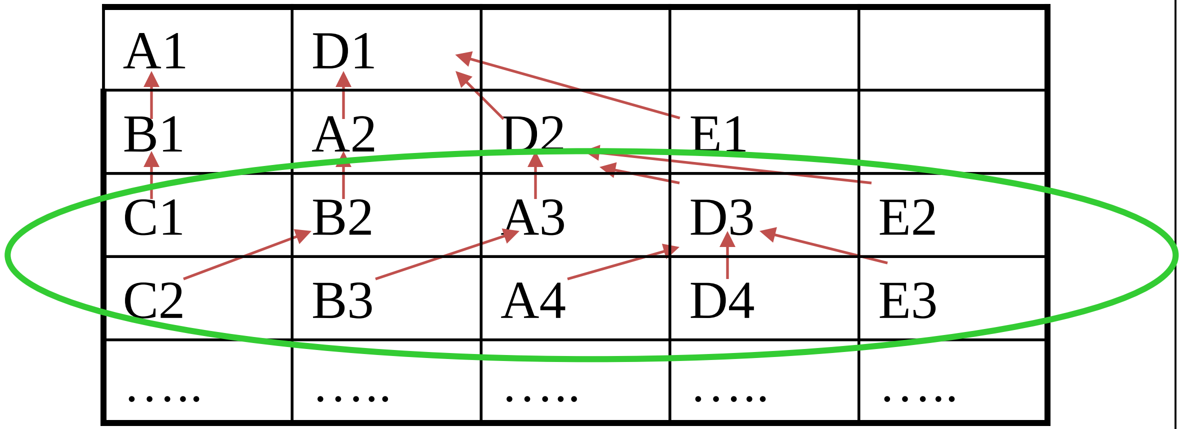
- Currently no two instructions can be executed in parallel on a statically scheduled machine.
  - (With branch prediction A and E could be executed in parallel)
- On a dynamically scheduled machine could execute instructions from different iterations at once.

What would OoO do?

```
r1=MEM[r2+0]    //A
r1=r1*2         //B
MEM[r2+0]=r1    //C
r2=r2+4         //D
bne r2 r3 Loop  //E
```

Let A1 indicate the execution of A in the first iteration of the loop.

- A perfect, dynamically scheduled, speculative computer would find the following:

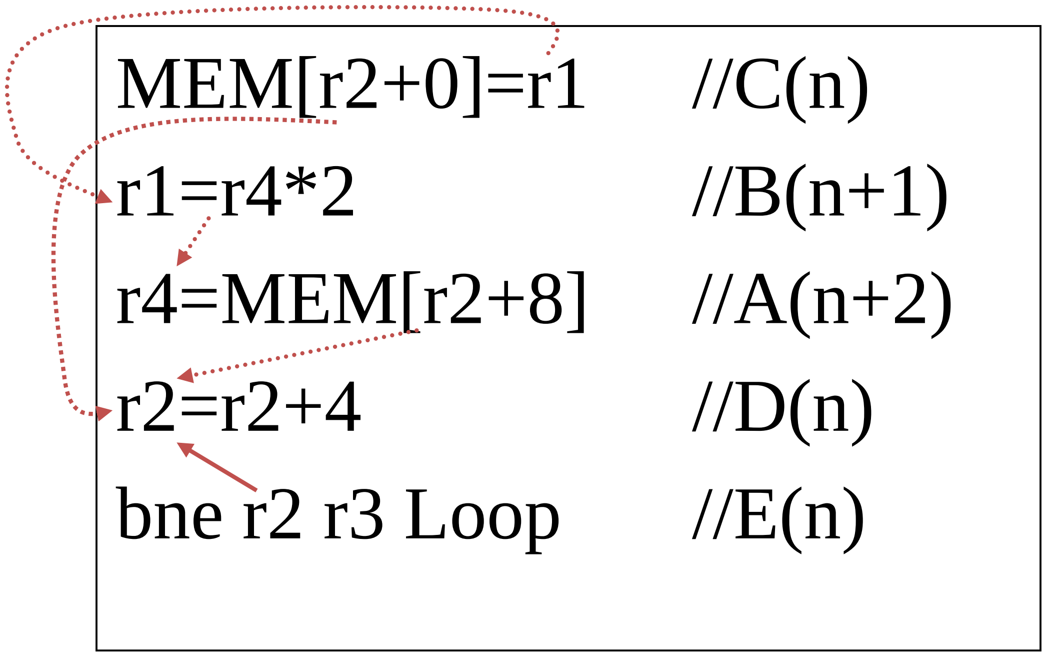


A1	D1			
B1	A2	D2	E1	
C1	B2	A3	D3	E2
C2	B3	A4	D4	E3
…..	…..	…..	…..	…..

Software Pipeline

```
r1=MEM[r2+0]   //A
r1=r1*2        //B
MEM[r2+0]=r1   //C
r2=r2+4        //D
bne r2 r3 Loop //E
```

- With "software pipelining" we can do the same thing in software.



```
MEM[r2+0]=r1      //C(n)
r1=r4*2           //B(n+1)
r4=MEM[r2+8]      //A(n+2)
r2=r2+4           //D(n)
bne r2 r3 Loop    //E(n)
```

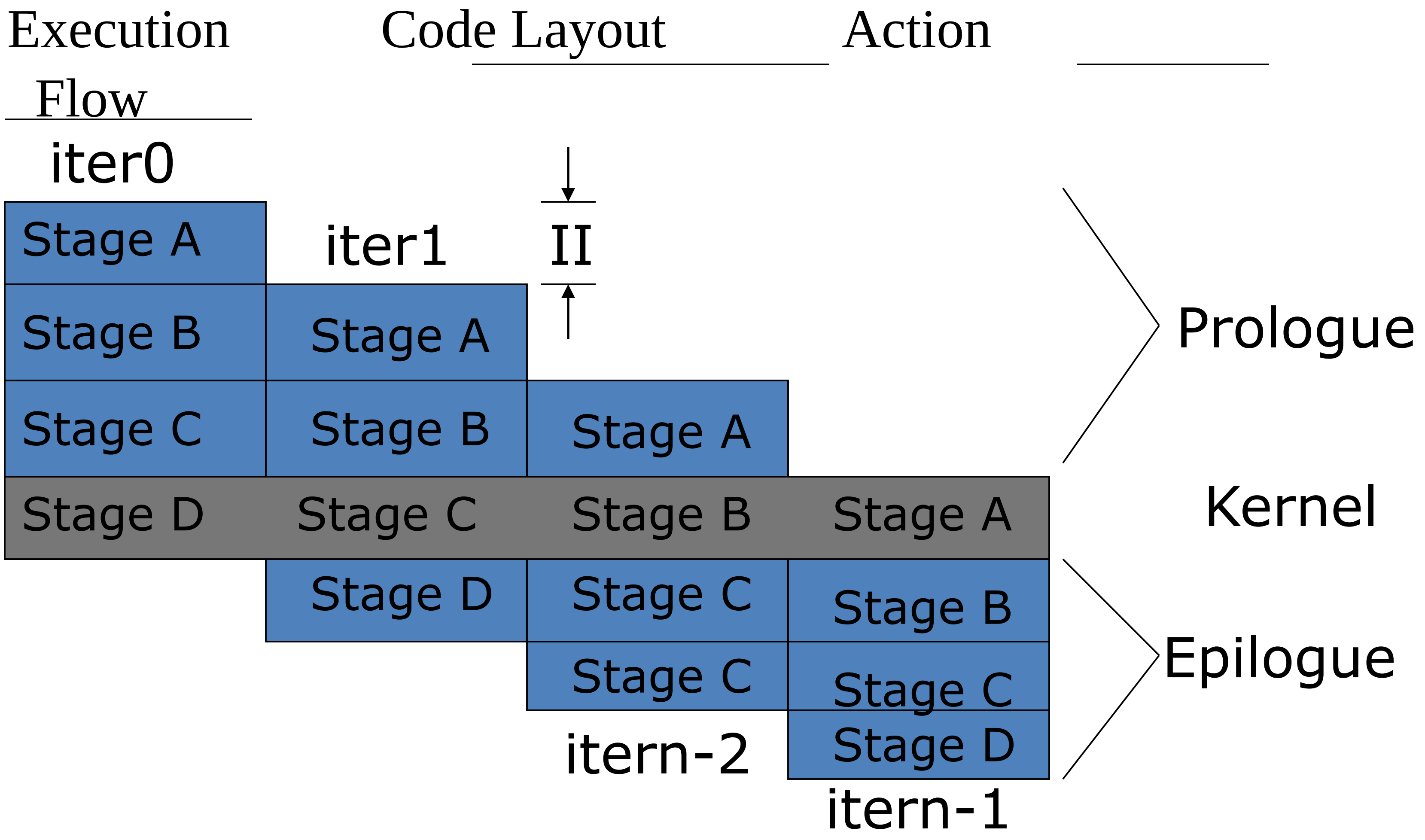
What problems could arise?

- "Speculative load" might cause an exception.
- Latency of load could be too slow.

Prolog and epilog

```
        r3=r3-8            // Needed to check legal!
        r4=MEM[r2+0]         //A(1)
        r1=r4*2              //B(1)
        r4=MEM[r2+4]         //A(2)
Loop:   MEM[r2+0]=r1         //C(n)
        r1=r4*2              //B(n+1)
        r4=MEM[r2+8]         //A(n+2)
        r2=r2+4              //D(n)
        bne r2 r3 Loop       //E(n)
        MEM[r2+0]=r1         // C(x-1)
        r1=r4*2              // B(x)
        MEM[r2+0]=r1         // C(x)
        r3=r3+8              // Could have used tmp var.
```

Software Pipelining example



Example, just to be sure.

```
        r4=MEM[r2+0]     //A1
        r1=r4*2          //B1
        r4=MEM[r2+4]     //A2
Loop:   MEM[r2+0]=r1     //C(n)
        r1=r4*2          //B(n+1)
        r4=MEM[r2+8]     //A(n+2)
        r2=r2+4          //D(n)
        bne r2 r3 Loop   //E(n)
```

ADDR	DATA
12	55
16	23
20	19
24	-5

R2=12, r3=28

R4=________ R1= ________

Next step

```
r1=MEM[r2+0]   //A
r1=r1*2        //B
MEM[r2+0]=r1   //C
r2=r2+4        //D
bne r2 r3 Loop //E
```

- Parallel execution
  - It isn't clear how D and E of any iteration can be executed in parallel on a statically scheduled machine
- What if load latency is too long?
  - Will be stalling a lot…
  - Fix by unrolling loop some.

NEXT…

- Let’s now jump from Software Pipelining to IA-64.
  - We will come back to Software Pipelining in the context of IA-64…
  - We will redo the IA-64 stuff from the start for next lecture
    - Not sure how far I’ll get into it today.

IA-64

- 128 64-bit registers
  - Use a register window similarish to SPARC
- 128 82 bit fp registers
- 64 1 bit predicate registers
- 8 64-bit branch target registers

Explicit Parallelism

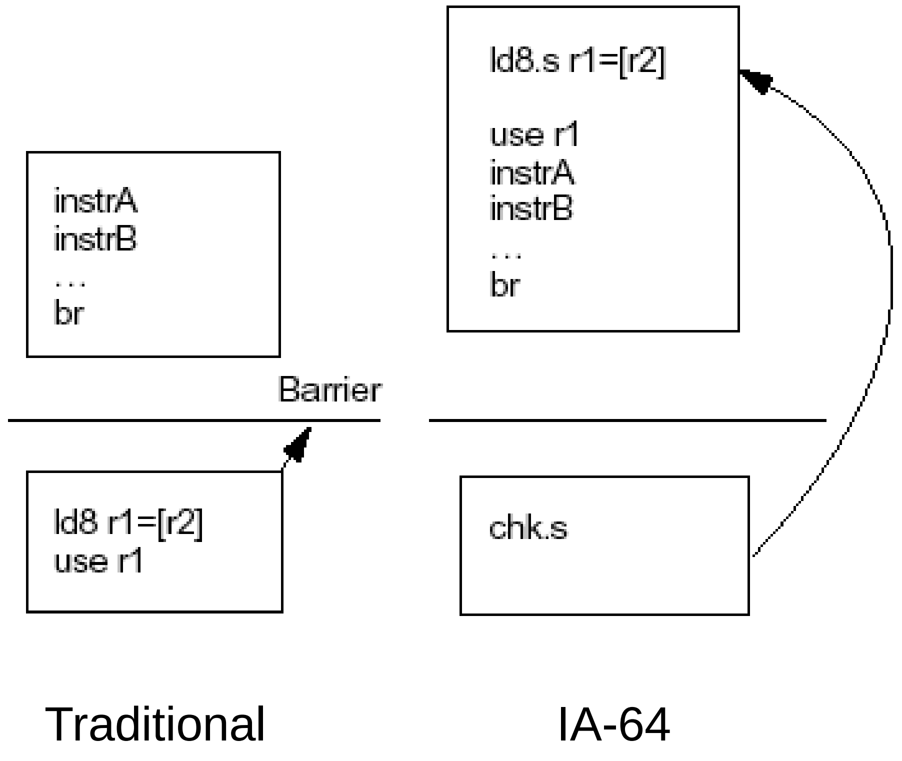
- Groups
  - Instructions which *could* be executed in parallel if hardware resources available.
- Bundle
  - Code format. 3 instructions fit into a 128-bit bundle.
  - 5 bits of template, 41*3 bits of instruction.
    - Template specifies what execution units each instruction requires.

Instructions

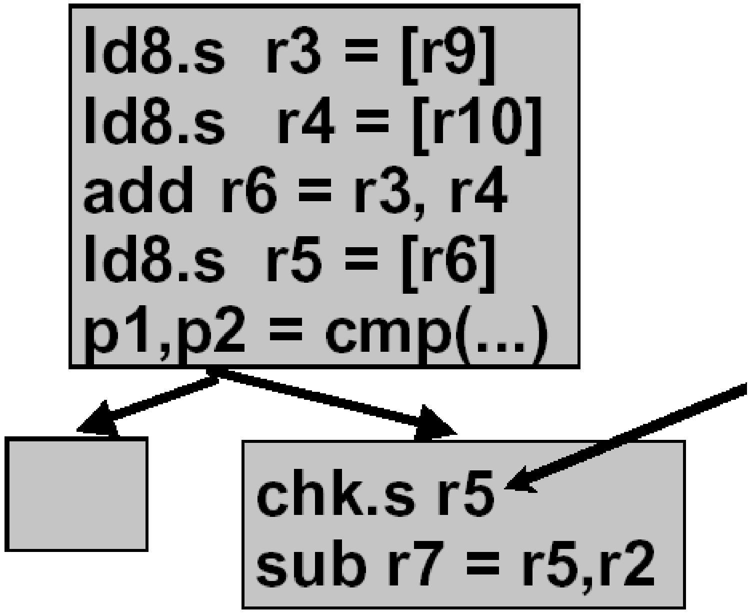
- 41 bits
  - 4 high order specify opcode (combined with template for bundle)
  - 6 low order bits specify predicate register number.
- Every instruction is predicated!
- Also NaT bits are used to handle speculated exceptions.

Speculative Load

- Load instruction (ld.s) can be moved outside of a basic block even if branch target is not known
- Speculative loads does not produce exception - it sets the NaT
- Check instruction (chk.s) will jump to fix-up code if NaT is set



Propagation of NaT



```
ld8.s  r3 = [r9]
ld8.s  r4 = [r10]
add r6 = r3, r4
ld8.s  r5 = [r6]
p1,p2 = cmp(...)
```

```
chk.s r5
sub r7 = r5,r2
```

Only single check required

NaT[reg] = NaT bit of reg

- IF (NaT[r3] || NaT[r4]) THEN set NaT[r6]
- IF (NaT[r6]) THEN set NaT[r5]
- Require check on NaT[r5] only since the NaT is inherited
- Reduce number of checks
- Fix-up will execute the entire chain

Advanced loads

- ld.a – Advanced load
  - Performs the load, puts it into the “ALAT”
    - If any following store writes to the same address, this is noted with a single bit.
    - When a ld.c is executed, if that bit is set, we refetch.
    - When chk.a is executed, if bit is set, fix up code is run. (Useful if load result already used.)
    - Both also cause any deferred exception to occur.

Software pipelining on IA-64

- Lots of tricks
  - Rotating registers
  - Special counters
- Often don’t need Prologue and Epilog.
  - Special counters and prediction lets us only execute those instructions we need to.