

COM515 Advanced Computer Architecture

Lecture 3. ILP (Instruction-Level Parallelism)

Prof. Taeweon Suh

Computer Science Education

Korea University

ILP

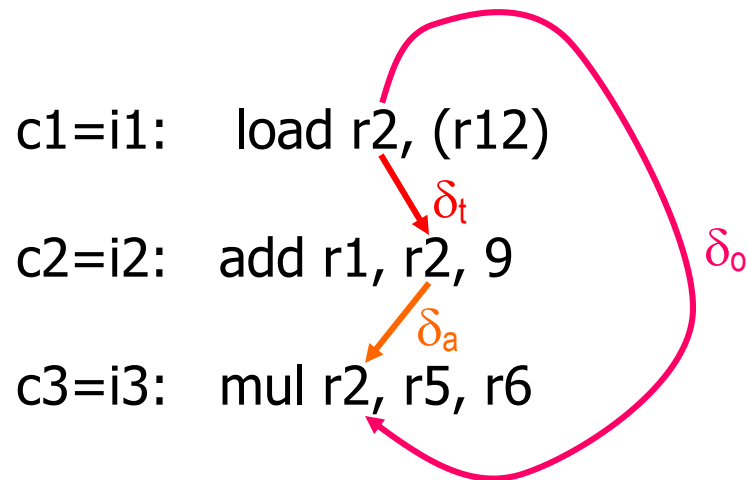
- Fine-grained parallelism
- All processors since about 1985 use pipelining to overlap the execution of instruction and improve performance
 - This potential overlap among instructions is called instruction-level parallelism (ILP), since the instructions can be evaluated in parallel
 - ILP is a measure of how many of the operations in a computer program can be performed simultaneously - Wikipedia
- There are 2 largely separable approaches to exploiting ILP
 - Hardware-based approach relies on hardware to help discover and exploit parallelism dynamically
 - Software-based approach relies on software technology (compiler) to find parallelism statically
- Limited by
 - Data dependency
 - Control dependency

Dependence & Hazard

- A hazard is created whenever there is a dependence between instructions, and they are close enough that the overlap during execution would change the order of access to the operand involved in the dependence
 - Because of the dependence, we must preserve the what it called program order
- Data hazard
 - RAW (Read After Write) or True data dependence
 - WAW (Write After Write) or Output dependence
 - WAR (Write After Read) or Antidependence
 - RAR (Read After Read) is not a hazard

ILP Example

- True dependency forces “sequentiality”
- $ILP = 3/3 = 1$
- False dependency removed
- $ILP = 3/2 = 1.5$



i1: load r2, (r12)
i2: add r1, r2, 9
i3: mul r8, r5, r6



c1: load r2, (r12)
c2: add r1, r2, #9 mul r8, r5, r6

Window in Search of ILP

ILP = 1

$$R5 = 8(R6)$$

$$R7 = R5 - R4$$

$$R9 = R7 * R7$$

ILP = 1.5

$$R15 = 16(R6)$$

$$R17 = R15 - R14$$

$$R19 = R15 * R15$$

ILP = ?

Window in Search of ILP

$R5 = 8(R6)$

$R7 = R5 - R4$

$R9 = R7 * R7$

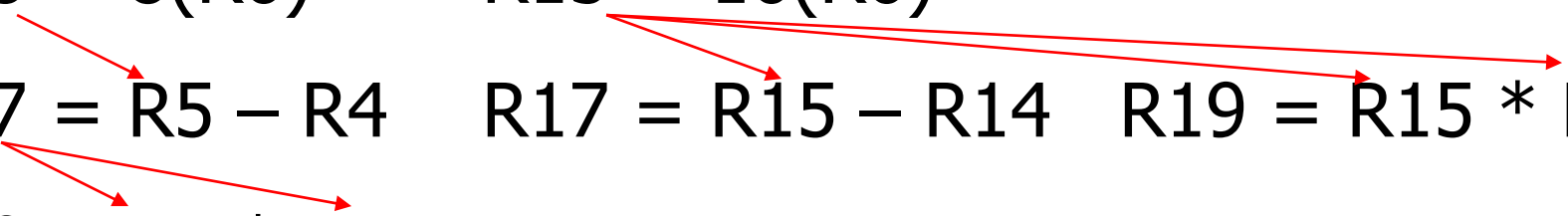
$R15 = 16(R6)$

$R17 = R15 - R14$

$R19 = R15 * R15$

Window in Search of ILP

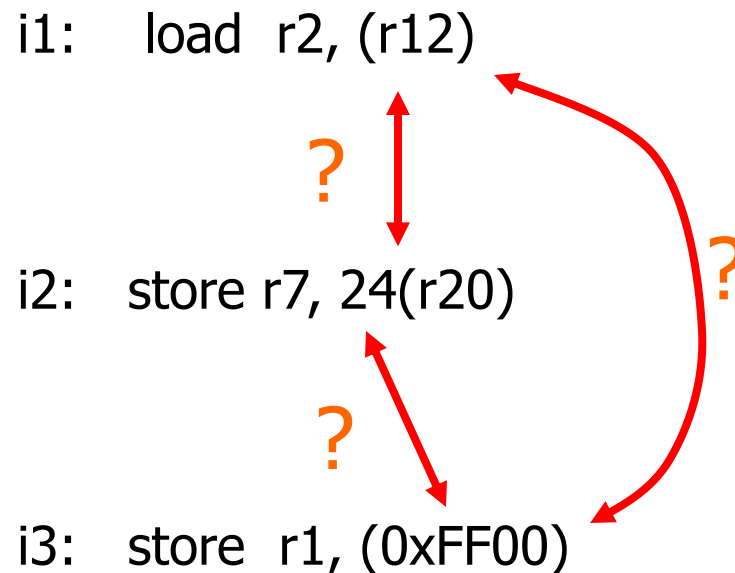
C1: $R5 = 8(R6)$ $R15 = 16(R6)$
C2: $R7 = R5 - R4$ $R17 = R15 - R14$ $R19 = R15 * R15$
C3: $R9 = R7 * R7$



- $ILP = 6/3 = 2$ better than 1 and 1.5
- Larger window gives more opportunities
- Who exploit the instruction window?
- But what limits the window?

Memory Dependency

- Ambiguous dependency also forces “sequentiality”
- To increase ILP, needs dynamic memory disambiguation mechanisms that are either safe or recoverable
- ILP could be 1, could be 3, depending on the actual dependence



ILP, Another Example

When only 4 registers available

$$R1 = 8(R0)$$

$$R3 = R1 - 5$$

$$R2 = R1 * R3$$

$$24(R0) = R2$$

$$R1 = 16(R0)$$

$$R3 = R1 - 5$$

$$R2 = R1 * R3$$

$$32(R0) = R2$$

ILP =

ILP, Another Example

When more registers (or register renaming) available

$$R1 = 8(R0)$$

$$R3 = R1 - 5$$

$$R2 = R1 * R3$$

$$24(R0) = R2$$

$$R5 = 16(R0)$$

$$R6 = R5 - 5$$

$$R2 = R5 * R6$$

$$32(R0) = R2$$

ILP =

Basic Block

- A straight line code sequence with no branches
- For typical MIPS programs, the average dynamic branch frequency is often between 15% and 25%
 - There are only 3 to 6 instructions between a pair of branches
 - But, these instructions are likely to depend on one another, the amount of overlap within a basic block is likely to be less than the average basic block size
- To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks

Basic Blocks

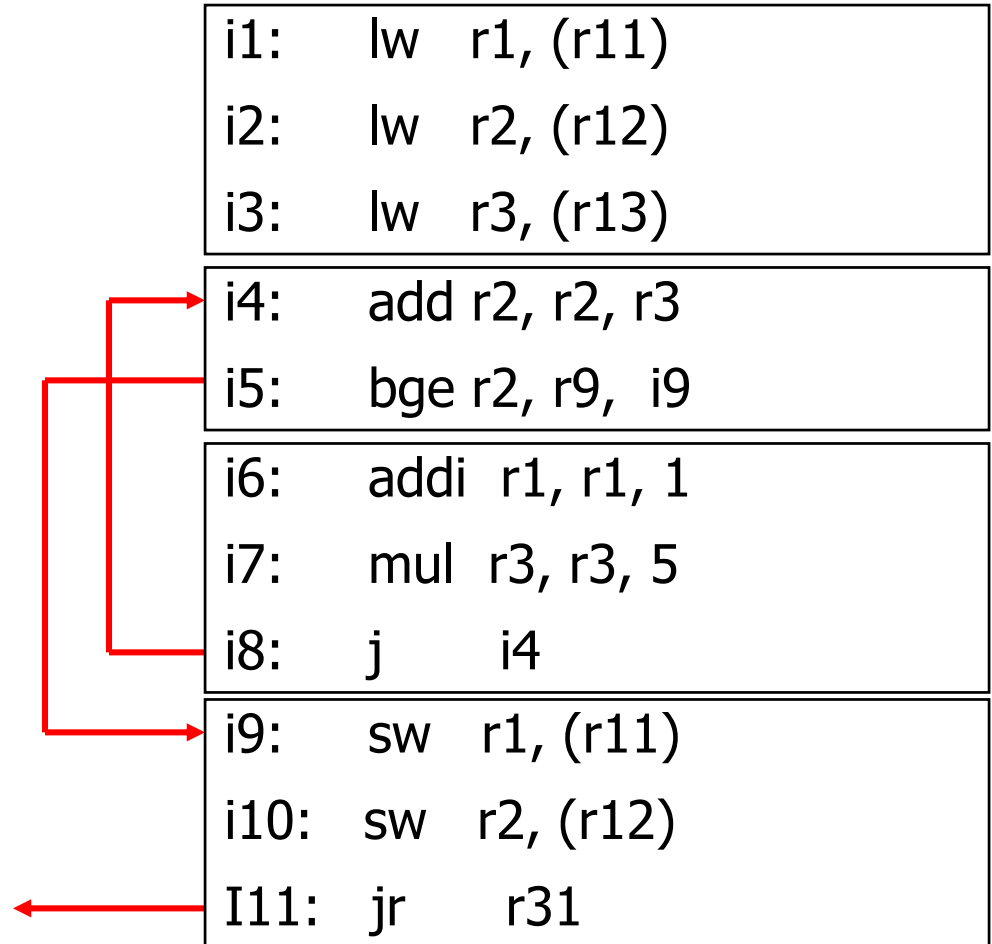
```
a = array[i];  
b = array[j];  
c = array[k];  
d = b + c;  
while (d < t) {  
    a++;  
    c *= 5;  
    d = b + c;  
}  
array[i] = a;  
array[j] = d;
```



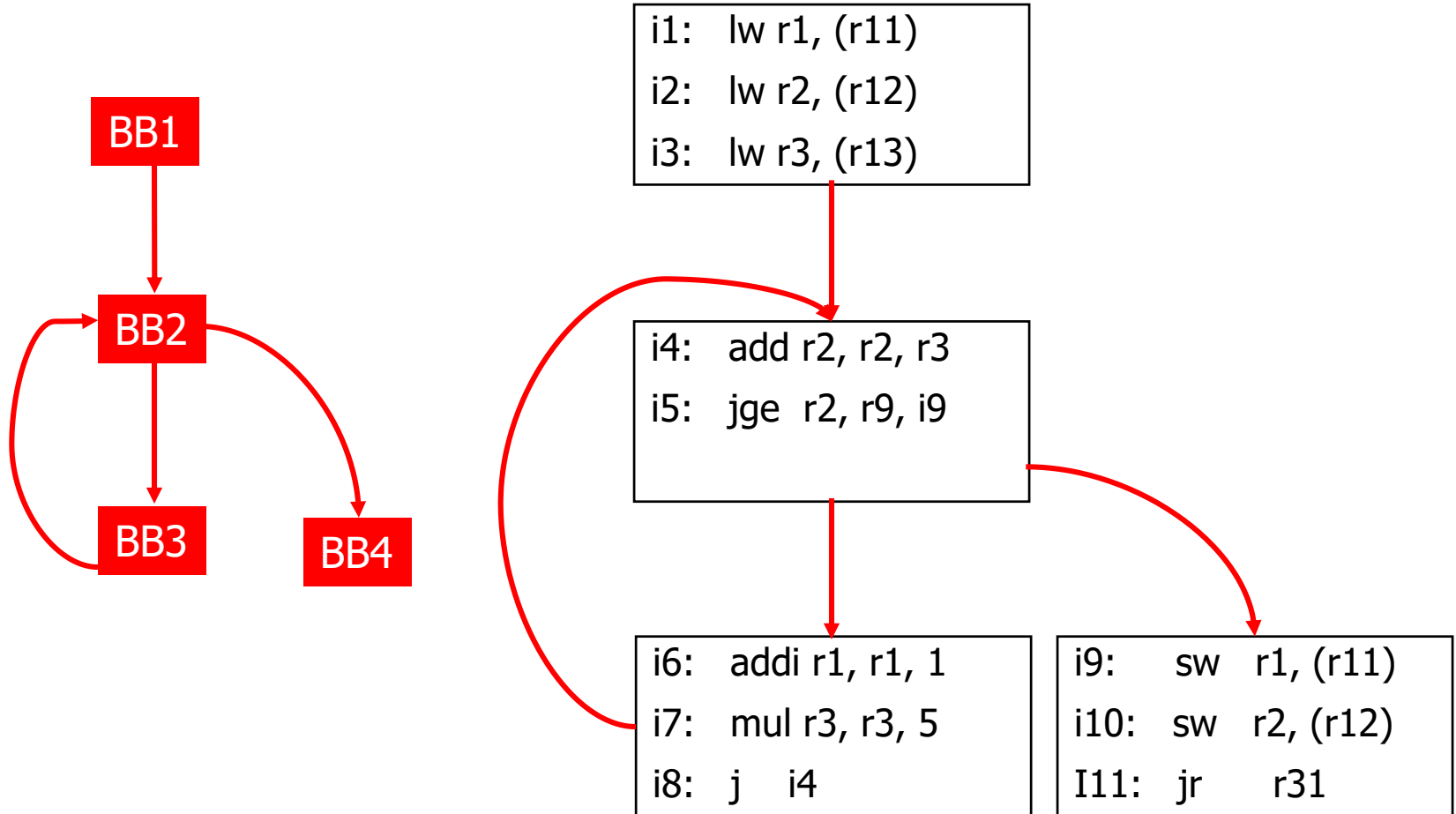
```
i1:  lw  r1, (r11)  
i2:  lw  r2, (r12)  
i3:  lw  r3, (r13)  
i4:  add r2, r2, r3  
i5:  bge r2, r9, i9  
i6:  addi r1, r1, 1  
i7:  mul r3, r3, 5  
i8:  j    i4  
i9:  sw  r1, (r11)  
i10: sw  r2, (r12)  
I11: jr   r31
```

Basic Blocks

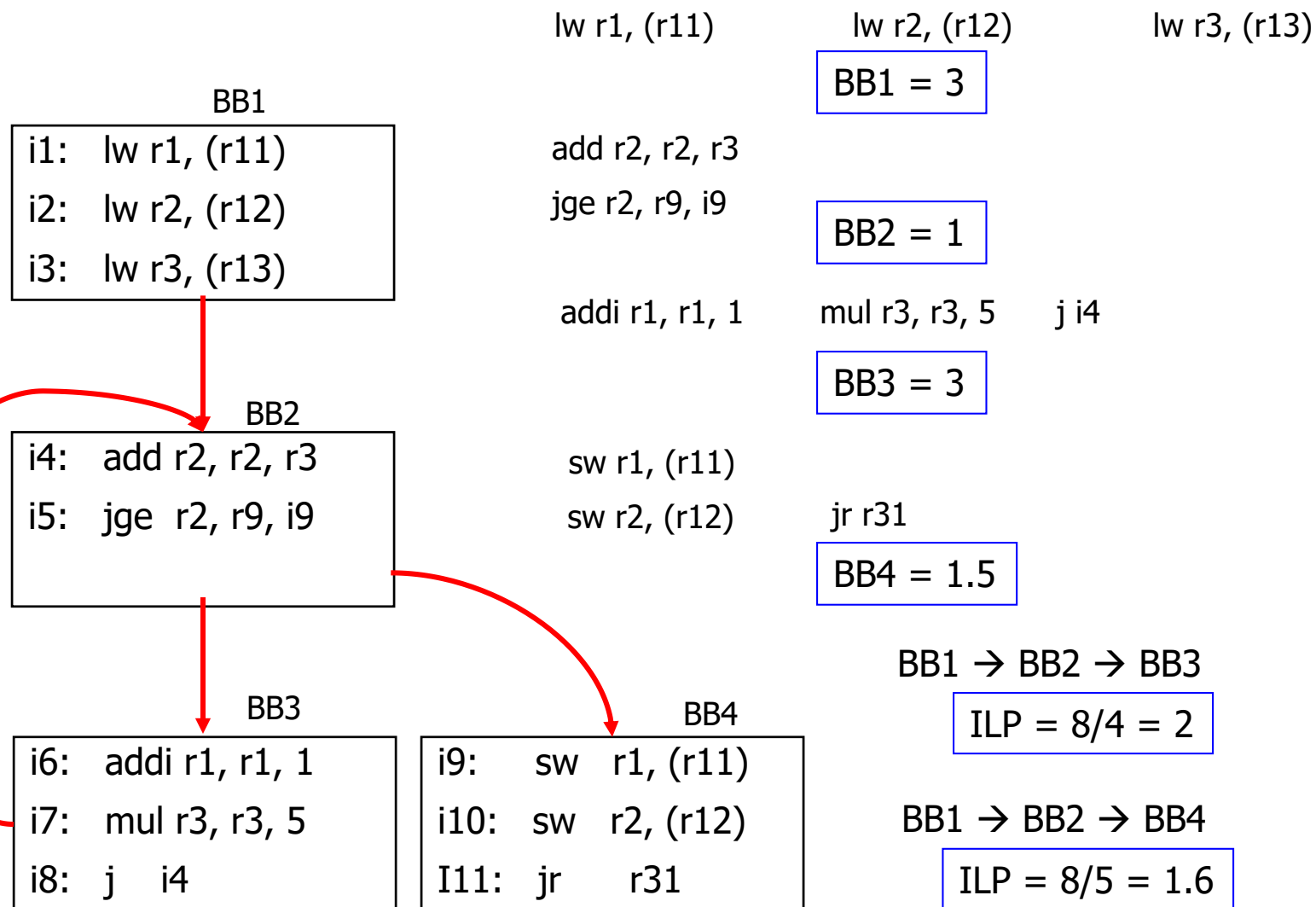
```
a = array[i];  
b = array[j];  
c = array[k];  
d = b + c;  
while (d < t) {  
    a++;  
    c *= 5;  
    d = b + c;  
}  
array[i] = a;  
array[j] = d;
```



Control Flow Graph



ILP (without Speculation)



ILP (with Speculation, No Control Dependence)

BB1 → BB2 → BB3

BB1

```
i1: lw r1, (r11)
i2: lw r2, (r12)
i3: lw r3, (r13)
```

```
lw r1, (r11)
add r2, r2, r3
jge r2, r9, i9
```

```
lw r2, (r12)
addi r1, r1, 1
j i4
```

```
lw r3, (r13)
mul r3, r3, 5
```

$$\text{ILP} = 8/3 = 2.67$$

BB2

```
i4: add r2, r2, r3
i5: jge r2, r9, i9
```

```
lw r1, (r11)
add r2, r2, r3
jge r2, r9, i9
```

BB1 → BB2 → BB4

```
lw r2, (r12)
sw r1, (r11)
sw r2, (r12)
```

```
lw r3, (r13)
jr r31
```

$$\text{ILP} = 8/3 = 2.67$$

BB3

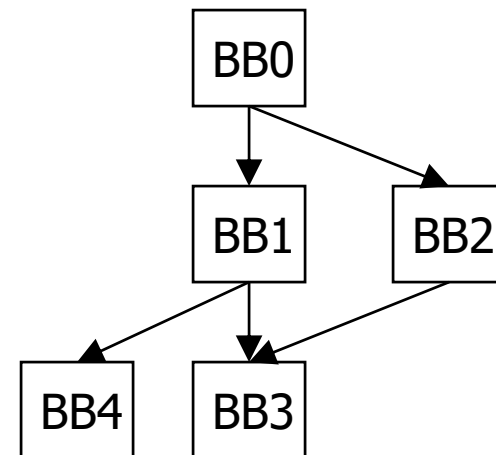
```
i6: addi r1, r1, 1
i7: mul r3, r3, 5
i8: j i4
```

BB4

```
i9: sw r1, (r11)
i10: sw r2, (r12)
i11: jr r31
```

Flynn's Bottleneck

- ILP ≈ 1.86 ☹
 - Programs on IBM 7090
 - ILP exploited **within basic blocks**
- [Riseman & Foster'72]
 - Breaking control dependency
 - A perfect machine model
 - Benchmark includes numerical programs, assembler and compiler



passed conditional jumps	0 jump	1 jump	2 jumps	8 jumps	32 jumps	128 jumps	∞ jumps
Average ILP	1.72	2.72	3.62	7.21	14.8	24.2	51.2



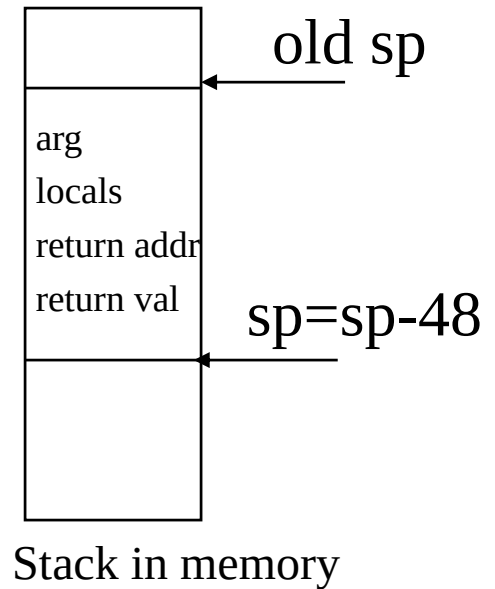
David Wall (DEC) 1993

- Evaluating effects of microarchitecture on ILP
- OOO with 2K instruction window, 64-wide, unit operation latency
- Peephole alias analysis (alias by instruction inspection) — inspecting instructions to see if any obvious independence between addresses
- Indirect jump predict —
 - Ring buffer (for procedure return): similar to return address stack
 - Table: last time prediction

models	branch predict	ind jump predict	reg renaming	alias analysis	ILP
Stupid	NO	NO	NO	NO	1.5 - 2
Poor	64b counter	NO	NO	peephole	2 - 3
Fair	2Kb ctr/gsh	16-addr ring no table	NO	Perfect	3 - 4
Good	16kb loc/gsh	16-addr ring 8-addr table	64 registers	perfect	5 - 8
Great	152 kb loc/gsh	2k-addr ring 2k-addr table	256	perfect	6 - 12
Superb	fanout 4, then 152kb loc/gsh	2k-addr ring 2k-addr table	256	perfect	8 - 15
Perfect	Perfect	Perfect	Perfect	perfect	18 - 50

Stack Pointer Impact

- Stack Pointer register dependency
 - True dependency upon each function call
 - Side effect of language abstraction
 - See execution profiles in the paper
- “Parallelism at a distance”
 - Example: **printf()**
 - One form of Thread-level parallelism



Removing Stack Pointer Dependency [Postiff'98]

Benchmark	IPC No Renaming	IPC Register Renaming	IPC Memory Renaming	IPC r29 Removed	IPC 10K Window
compress95	3.12	26.25	73.88	226.33	18.89
cc1	3.61	39.79	41.63	239.96	86.45
go	2.50	49.15	53.77	141.46	70.71
jpeg	2.41	55.47	93.60	94.11	52.94
li	3.56	19.60	19.61	81.45	27.70
m88ksim	2.76	19.93	62.06	363.26	20.50
perl	3.47	82.01	127.57	153.05	128.84
vortex	4.57	26.26	26.27	271.97	92.04
applu	2.82	106.65	2037.61	2076.06	78.67
apsi	3.6	54.89	183.44	1224.86	79.56
fpppp	3.33	103.62	774.13	1837.96	134.62
hydro2d	3.09	144.80	147.67	242.08	52.14
mgrid	3.34	1876.11	3933.03	4003.44	286.48
su2cor	3.22	38.21	34.81	55.56	47.60
swim	3.10	112.08	112.08	275.21	89.15
tomcatv	3.61	32.85	61.47	119.67	58.91
turb3d	3.42	370.98	482.24	3652.46	0
wave5	3.25	29.28	35.71	35.71	0

↑
\$sp effect

Exploiting ILP

- Hardware
 - Control speculation (control)
 - Dynamic Scheduling (data)
 - Register Renaming (data)
 - Dynamic memory disambiguation (data)
- Software  Many embedded system designers chose this
 - (Sophisticated) program analysis
 - Predication or conditional instruction (control)
 - Better register allocation (data)
 - Memory Disambiguation by compiler (data)

Other Parallelisms

- SIMD (Single instruction, Multiple data)
 - Each register as a collection of smaller data
- Vector processing
 - e.g. VECTOR ADD: add long streams of data
 - Good for very regular code containing long vectors
 - Bad for irregular codes and short vectors
- Multithreading and Multiprocessing (or Multi-core)
 - Cycle interleaving
 - Block interleaving
 - High performance embedded's option (e.g., packet processing)
- Simultaneous Multithreading (SMT): *Hyper-threading*
 - Separate contexts, shared other microarchitecture modules