Static Instruction Level Parallelism

- Basics on Static ILP
- Pipeline scheduling and loop unrolling
- Static multiple issue: the VLIW approach
- Predicated instructions
- *IA-64 architecture: Itanium II (skipped)*
- Trimedia TM32

Static ILP

- So far, we have discussed the exploitation of parallelism available among instructions during the execution.
- Pipeline dynamic scheduling, register renaming, branch prediction, speculation, multiple issue, are all run time techniques.
- This is why all techniques that exploit the available parallelism in a program in this mode are referred to as "dynamic ILP".

Static ILP

- Compiler can play a major role in exploiting ILP, by reordering the generated instructions, so that the pipeline and the functional units will be bettered utilized when executing the instructions.
- In so doing, the compiler must preserve the intended behaviour of the program (semantics), as designed by the programmer.
- These techniques try to leverage on available parallelism of instructions *before they enter execution*, and therefore they are called **Static ILP.**

Static ILP

- Static ILP is most important in *embedded processors*, that must be inexpensive and energy saving: they simpply cannot allow for the huge number of transistors required for dynamic ILP.
- Static ILP is also used in the purely RISC **IA-64** ISA (architecture + instruction set) developed by Intel and HP since 2000, at the heart of Itanium and Itanium II processors.
- Actually, static and dynamic ILP do not split sharply, and many modern architectures use some combination of both techniques.

Static ILP: the concept

- Static ILP concept: keeping the pipeline working (avoiding stalls) by looking (**at compile time**) for sequences of un-correlated instructions that can overlap during execution in the pipeline.

- To prevent being stalled, an instruction B dependent on instruction A must be separated from A by an amount of clock cycle at least equal to the number of cycles necessary for A to produce its result.

- The **compiler** can (and, if, possible, must) place between A and B other instructions, that of course should not introduce further dependencies.

Static ILP: the concept

- The *actual* possibility for a compiler to perform this type of "instruction scheduling" depends:
  - on the amount of ILP available in the program being compiled
  - on the length of the pipeline, on the type and number of available functional units, and on the execution time of the various instructions
  - **on the compiler knowing all these data for the CPU it is generating code for !!**
- Two CPUs with the same ISA but different microarchitectures (a different datapath, a set of different FUs and possibly a different number of pipeline stages), can yield different performances on the code generated by a compiler that exploits Static ILP: the compiler takes care of each specific microarchitecture. What about dynamic ILP ?

Basic compiler techniques

- Let us consider a compiler generating code for a 5-stage pipeline CPU with the following characteristics:
  - the FUs are themselves pipelined, and a new instruction can be launched at each new clock cycle
  - Branches take two clock cycles
  - Before instruction B can use the result produced by instruction A, the following clock cycles must elapse (Hennessy-Patterson, Fig. 4.1):

(A) instruction producing result	(B) instruction using result	latency (clock cycles)
FP ALU op	another FP ALU op	3
FP ALU op	Store double	2
double ALU op	branch	1
Load double	FP ALU op	1

Basic techniques: pipeline scheduling

- Let us consider the following *for*, that adds a scalar to all elements of a 1000-element array:

```
for ( i = 1000; i > 0; i = i-1 )
        x[i] = x[i] + s;
```

- and its MIPS code translation (assuming that R1 and R2 have been pre-computed):

```
LOOP: LD     F0, 0 (R1)       // F0 = array element
      FADD   F4, F0, F2       // scalar is in F2
      SD     F4, 0 (R1)       // store result
      DADD   R1, R1, #-8      // pointer to array is in R1 (DW)
      BNE    R1, R2, LOOP     // suppose R2 pre-computed
```

Basic techniques: pipeline scheduling

- Here is an "execution" of the *for* loop with no scheduling:

			clock cycle issued
LOOP:	LD	F0, 0 (R1)	1
	nop		2
	FADD	F4, F0, F2	3
	stall		4
	stall		5
	SD	F4, 0 (R1)	6
	DADD	R1, R1, #-8	7
	stall		8
	BNE	R1, R2, LOOP	9
	stall		10

No branch prediction

- The three RAW hazards cause stalls in the pipeline, and the *for* loop requires 10 clock cycles for a single iteration.

Basic techniques: pipeline scheduling

- A "smart" compiler, with a notion of FUs latencies (previous table) and knowing that delayed branch can be enforced, could do some **pipeline scheduling**: namely, instructions reordering, so as to reduce the number of stalls:

```
                                    clock cycle issued
LOOP:  LD     F0, 0 (R1)                  1
       DADD   R1, R1, #-8                 2   // one stall "filled"
       FADD   F4, F0, F2                  3
       stall                              4
       BNE    R1, R2, LOOP                5   // delayed branch
       SD     F4, 8 (R1)                  6   // altered &
                                                 interchanged
                                                 with DADD
```

Basic techniques: pipeline scheduling

```
LOOP:   1  LD      F0, 0 (R1)
        2  DADD    R1, R1, #-8
        3  FADD    F4, F0, F2
        4  stall
        5  BNE     R1, R2, LOOP
        6  SD      F4, 8 (R1)
```

- Which are the changes?
  - DADD moved in the second slot eliminates stalls (and associated wasted clock cycles) between LD and FADD, and between DADD and BNE
  - moving SD after BNE eliminates the 1-cycle stall due to BNE and decreases by 1 the stall between FADD and SD
  - since SD and DADD have been swapped, the offset in SD must be 8 instead of 0
  - This type of scheduling is only possible if the compiler knows the latency (in clock cycles) of each generated instruction (that is, if it knows the details of the hardware organization of the architecture for which it is generating code)

Basic techniques: loop unrolling

- The scheduled code is much better, but one can note that the 6 clock cycles necessary for one iteration in the loop are used as follows:
  - 3 to work on the array's elements (LD,ADD,SD)
  - 3 to manage *loop overhead* (DADD, BNE, *stall*)
- An alternative approach to pipeline scheduling is increasing the number of instructions that actually work on the array, with respect to those used for loop management. How?
- Instructions from consecutive iterations can be merged into a single "fat" iteration, managed by a single control point (a unique BNE). This is a form of **static loop unrolling**.

Basic techniques: loop unrolling

As an instance, if the number of iterations N≈0 mod(4), the original loop can be unrolled 4 times:

```
LD      F0, 0 (R1)
FADD    F4, F0, F2
SD      F4, 0 (R1)       // drop DADD & BNE
LD      F6, -8 (R1)
FADD    F8, F6, F2
SD      F8, -8 (R1)      // drop DADD & BNE
LD      F10, -16 (R1)
FADD    F12, F10, F2
SD      F12, -16 (R1)    // drop DADD & BNE
LD      F14, -24 (R1)
FADD    F16, F14, F2
SD      F16, -24 (R1)
DADD    R1, R1, #-32
BNE     R1, R2, LOOP
```

Basic techniques: loop unrolling

- Let us not consider what the compiler has done about the registers (a form of "renaming").
- If the code is executed as it is, its execution requires 28 clock cycles, because of instructions latencies, that induce some stalls in the pipeline. From the latency table we have:
  - 1-cycle stall between any LD and the associated FADD
  - 2-cycle stall between any FADD and the associated SD
  - 1-cycle stall between the DADD and the BNE
- as a result, each for loop takes 7 clock cycles on average: better than the first solution, but worse than with the pipeline scheduling just considered.

Basic techniques: loop unrolling & pipeline scheduling

- Branch elimination through loop unrolling allows to put into a single "large loop" instructions that belong in different iterations of the for loop
- Furthermore, register renaming has increased the number of independent instructions within the "large loop", that can be thus re-orderd in a profitable way.
- That is to say, the code can be optimized by combining pipeline scheduling with loop unrolling.

Loop unrolling & pipeline scheduling

```
LD      F0, 0 (R1)
LD      F6, -8 (R1)
LD      F10, -16 (R1)
LD      F14, -24 (R1)
FADD    F4, F0, F2
FADD    F8, F6, F2
FADD    F12, F10, F2
FADD    F16, F14, F2
SD      F4, 0 (R1)
SD      F8, -8 (R1)
DADD    R1, R1, #-32
SD      F12, 16 (R1)
BNE     R1, R2, LOOP
SD      F16, 8 (R1)
```

unrolled instructions (loop unrolling) can be re-ordered(pipeline scheduling) to eliminate of stalls

moving these two instructions allows to eliminate the last two stalls

// 8 – 32 = -24 (?)

Loop unrolling & pipeline scheduling

- A careful inspection of the code allows to verify that it is equivalent to the 4 iteration of the for loop.
- 4 iterations are carried out in 14 clock cycles, that is 3,5 clock cycles per iteration, against 10 necessary before the “compilation”

Loop unrolling & pipeline scheduling implementation details

- The primary purpose of unrolling is to identify “linear segments of code”, that is sequences of instructions with no intervening branch – this is a basic compilation optimization
- Instructions scheduling is easier within “linear segments of code”
- Unrolling is not always “possible”, and it has also some drawbacks, so it must be used when it is worth doing

Loop unrolling & pipeline scheduling implementation details

- Conditions for unrolling
  - No loop-carried dependency
  - enough resources (registers) available
- Drawbacks of unrolling
  - longer code “footprint” (executable)
  - shortage of RAM in embedded processors

Loop unrolling & pipeline scheduling implementation details

- Loop dependency analysis

- There can be three kinds of dependencies between statements:

flow dependence

- Also, **true dependence** or **definition-use dependence**.

```
(i)      X   := ···
             ......
(j)      ··· := X
```

- Statement (i) generates (**defines**) a value which is used by statement (j). We write (i) $\longrightarrow$ (j).

anti-dependence

- Also, **use-definition dependence**.

```
(i)      ··· := X
             ......
(j)      X   := ···
```


From CS553 Principles of Compilation, Univ. Arizona, Copyright © 2011 C. Collberg

Loop unrolling & pipeline scheduling implementation details

- Loop dependency analysis

- Statement `(i)` uses a value overwritten by statement `(j)`. We write `(i)`$\longrightarrow\!\!\!\!\!\!+\;\,$`(j)`.

Output-dependence

- Also, **definition-definition dependence**.

```
(i)        X   := ···              .....
(j)        X   := ···
```

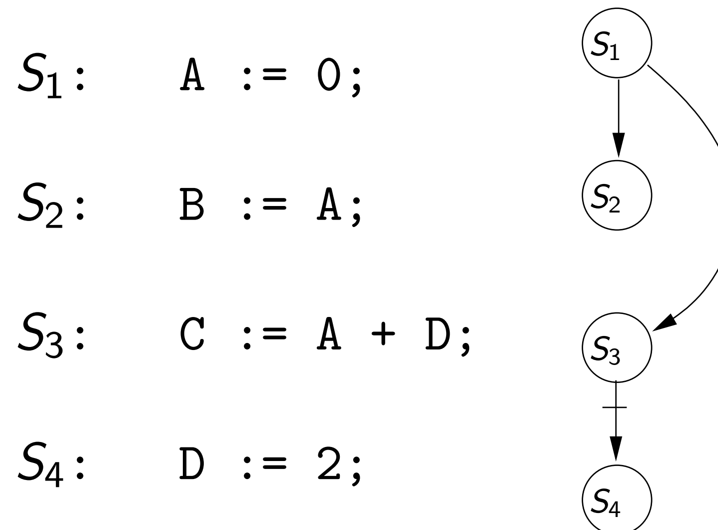
- Statements `(i)` and `(j)` both assign to (**define**) the same variable. We write `(i)`$\longrightarrow\!\!\!\!\!\!\circ\;\,$`(j)`.

- Regardless of the type of dependence, if statement `(j)` depends on `(i)`, then `(i)` has to be executed before `(j)`.

From CS553 Principles of Compilation, Univ. Arizona, Copyright © 2011 C. Collberg

Loop unrolling & pipeline scheduling implementation details

- Dependence graph



- In any program without loops, the dependence graph will be acyclic.
- Other common notations are

Flow	$\longrightarrow$	$\equiv$	δ	$\equiv$	δ^f	→T
Anti	$\not\longrightarrow$	$\equiv$	$\bar{\delta}$	$\equiv$	δ^a	→A
Output	$\multimap\!\!\rightarrow$	$\equiv$	δ°	$\equiv$	δ^o	→O

From CS553 Principles of Compilation, Univ. Arizona, Copyright © 2011 C. Collberg

Loop unrolling & pipeline scheduling implementation details

- Iteration Space: the set of *iteration vectors*

```
FOR i := 1 TO 3 DO
  FOR j := 1 TO 4 DO
    statement
  ENDFOR
ENDFOR
```

$\{\langle 1,1\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 1,4\rangle,$
$\langle 2,1\rangle, \langle 2,2\rangle, \langle 2,3\rangle, \langle 2,4\rangle,$
$\langle 3,1\rangle, \langle 3,2\rangle, \langle 3,3\rangle, \langle 3,4\rangle\}.$

```
FOR i := 1 TO 3 DO
  FOR j := 1 TO i+1 DO
    statement
  ENDFOR
ENDFOR
```

$\{\langle 1,1\rangle, \langle 1,2\rangle,$
$\langle 2,1\rangle, \langle 2,2\rangle, \langle 2,3\rangle,$
$\langle 3,1\rangle, \langle 3,2\rangle, \langle 3,3\rangle, \langle 3,4\rangle\}$

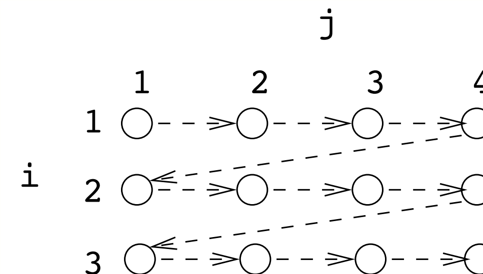
From CS553 Principles of Compilation, Univ. Arizona, Copyright © 2011 C. Collberg

Loop unrolling & pipeline scheduling implementation details

- Iteration space Traversal Graph (ITG): the order of traversal in the iteration space, that is a lexicographical ordering of the *iteration vectors*

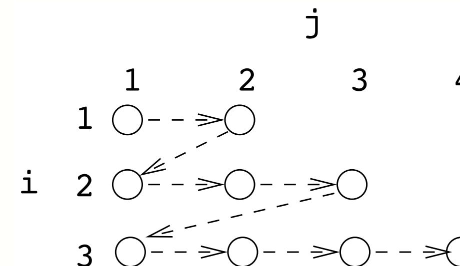
```
FOR i := 1 TO 3 DO
  FOR j := 1 TO 4 DO
    statement
  ENDFOR
ENDFOR
```

$\{\langle 1,1\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 1,4\rangle, \langle 2,1\rangle, \langle 2,2\rangle, \langle 2,3\rangle, \langle 2,4\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 3,3\rangle, \langle 3,4\rangle\}.$




```
FOR i := 1 TO 3 DO
  FOR j := 1 TO i+1 DO
    statement
  ENDFOR
ENDFOR
```

$\{\langle 1,1\rangle, \langle 1,2\rangle, \langle 2,1\rangle, \langle 2,2\rangle, \langle 2,3\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 3,3\rangle, \langle 3,4\rangle\}$



From CS553 Principles of Compilation, Univ. Arizona, Copyright © 2011 C. Collberg

Loop unrolling & pipeline scheduling implementation details

- Loop-carried vs Loop-independent dependences

  Loop-carried dependence exists across iterations, if the loop is removed, it no longer exists

  Loop-independent dependence exists within an iteration, if the loop is removed, it exists

Loop unrolling & pipeline scheduling implementation details

- Loop-carried vs Loop-independent dependences

```
for (i=1; i<n; i++) {
  S1: a[i] = a[i-1] + 1;
  S2: b[i] = a[i];
}

for (i=1; i<n; i++)
  for (j=1; j< n; j++)
    S3: a[i][j] = a[i][j-1] + 1;

for (i=1; i<n; i++)
  for (j=1; j< n; j++)
    S4: a[i][j] = a[i-1][j] + 1;
```

`S1[i]` →T `S1[i+1]`: loop-carried

`S1[i]` →T `S2[i]`: loop-independent

`S3[i,j]` →T `S3[i,j+1]`:

- loop-carried on **for** `j` loop
- no loop-carried dependence in **for** `i` loop

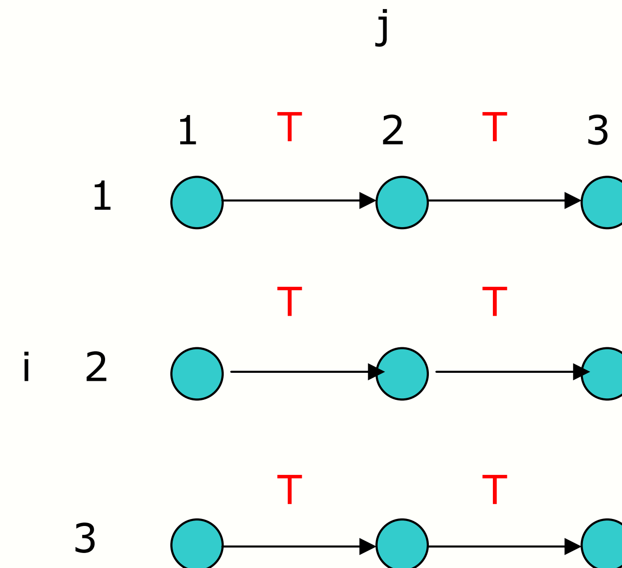
`S4[i,j]` →T `S4[i+1,j]`:

- no loop-carried dependence in **for** `j` loop
- loop-carried on **for** `i` loop

Loop unrolling & pipeline scheduling implementation details

- Loop-carried Dependence Graph (LDG): it shows True/Anti/Output dependences, each *node* is an *iteration vector*, each *directed edge* represents a *dependence*.

```
for (i=1; i<4; i++)
  for (j=1; j<4; j++)
    S3: a[i][j] = a[i][j-1] + 1;
```

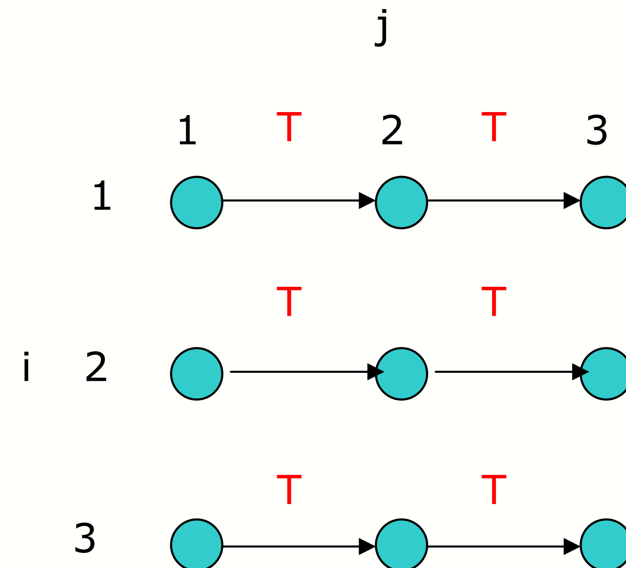


CSC/ECE 506 Lecture Notes, Copyright 2010 © F. Gehringer

Loop unrolling & pipeline scheduling implementation details

- Loop-carried Dependence Graph (LDG): it shows True/Anti/Output dependences, each *node* is an *iteration vector*, each *directed edge* represents a *dependence*.

```
for (i=1; i<4; i++)
  for (j=1; j<4; j++)
    S3: a[i][j] = a[i][j-1] + 1;
```



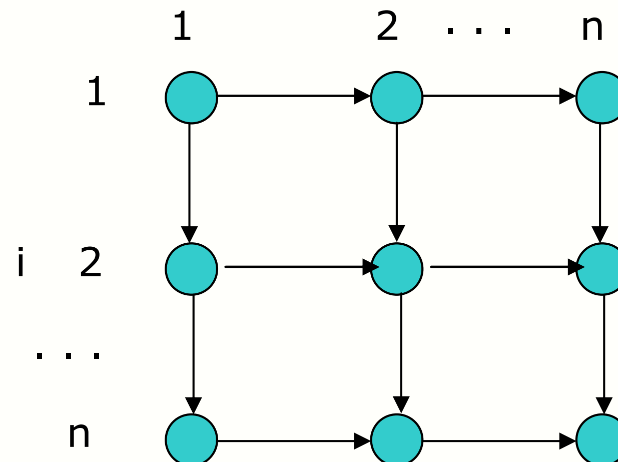
CSC/ECE 506 Lecture Notes, Copyright 2010 © F. Gehringer

Loop unrolling & pipeline scheduling implementation details

- Loop-carried Dependence Graph (LDG): it shows True/Anti/Output dependences, each *node* is an *iteration vector*, each *directed edge* represents a *dependence*.

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    S1: a[i][j] = a[i][j-1] + a[i][j+1] + a[i-1][j] + a[i+1][j];
```

- True dependences:
  - `S1[i,j]` →T `S1[i,j+1]`
  - `S1[i,j]` →T `S1[i+1,j]`
- Output dependences:
  - None
- Anti-dependences:
  - `S1[i,j]` →A `S1[i+1,j]`
  - `S1[i,j]` →A `S1[i,j+1]`



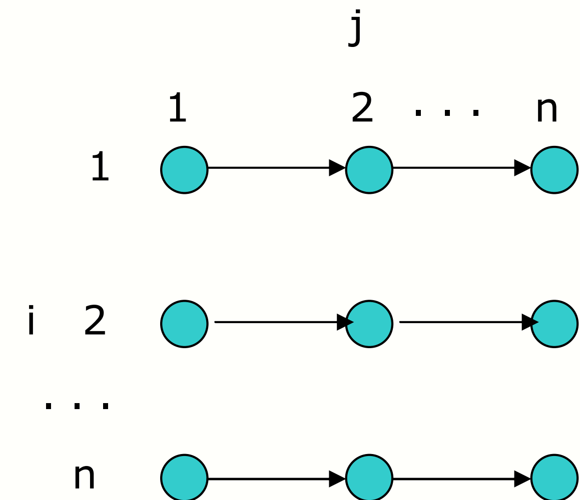
Note: each edge represents both true, and anti-dependences

Loop unrolling & pipeline scheduling implementation details

- Loop-carried Dependence Graph (LDG): it shows True/Anti/Output dependences, each *node* is an *iteration vector*, each *directed edge* represents a *dependence*.

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S2: a[i][j] = b[i][j] + c[i][j];
    S3: b[i][j] = a[i][j-1] * d[i][j];
  }
```

- True dependences:
  - **S2[i,j]** →T **S3[i,j+1]**
- Output dependences:
  - None
- Anti-dependences:
  - **S2[i,j]** →A **S3[i,j]** (loop-independent dependence)



Note: each edge represents only true dependences

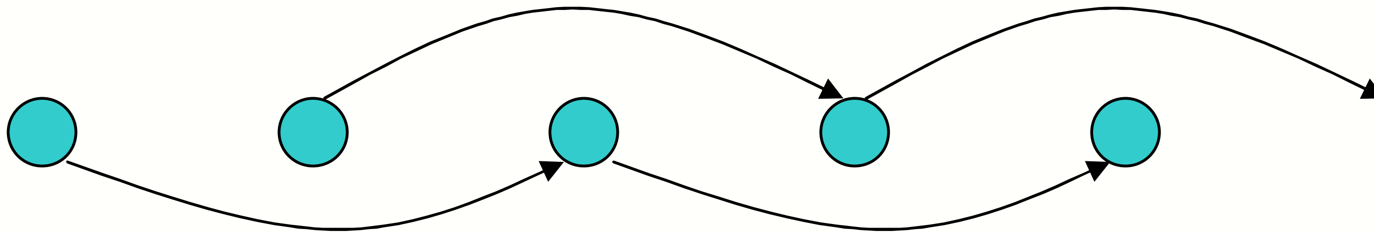
CSC/ECE 506 Lecture Notes, Copyright 2010 © F. Gehringer

Loop unrolling & pipeline scheduling implementation details

- Finding parallel tasks across iterations: identify all dependences, remove anti-dependences (renaming), identifying independent, disjoint subgraph in the LDG

```
for (i=2; i<=n; i++)
  S: a[i] = a[i-2];
```

LDG:

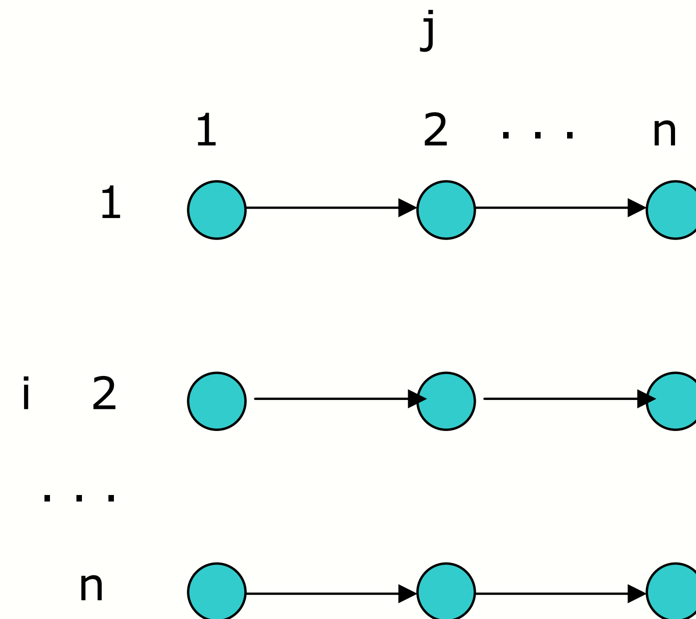


```
for (i=2; i<=n; i+=2)
  S: a[i] = a[i-2];
for (i=3; i<=n; i+=2)
  S: a[i] = a[i-2];
```

Loop unrolling & pipeline scheduling implementation details

- Finding parallel tasks across iterations: identify all dependences, remove anti-dependences (renaming), identifying independent, disjoint subgraph in the LDG

```
for (i=0; i<n; i++)
  for (j=0; j< n; j++)
    S3: a[i][j] = a[i][j-1] + 1;
```

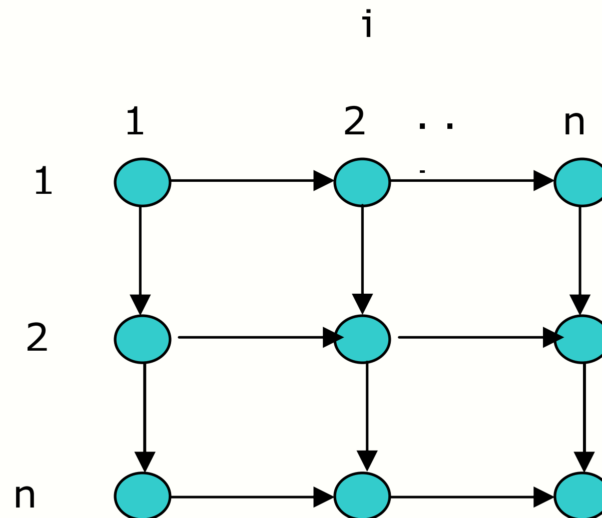


Loop unrolling & pipeline scheduling implementation details

- Finding parallel tasks across iterations: identify all dependences, remove anti-dependences (renaming), identifying independent, disjoint subgraph in the LDG

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    S1: a[i][j] = a[i][j-1] + a[i][j+1] + a[i-1][j] + a[i+1][j];
```

LDG

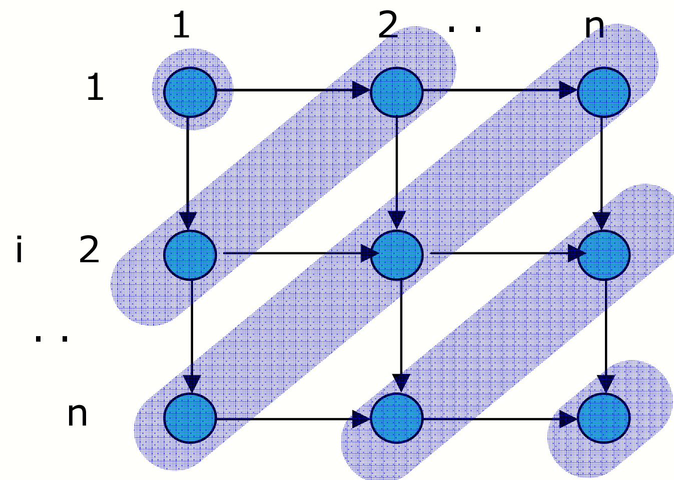


Note: each edge represents both true, and anti-dependences

Loop unrolling & pipeline scheduling implementation details

- Finding parallel tasks across iterations: identify all dependences, remove anti-dependences (renaming), identifying independent, disjoint subgraph in the LDG

In each anti-diagonal, the nodes are independent of each other



Note: each edge represents both true, and anti-dependences

CSC/ECE 506 Lecture Notes, Copyright 2010 © F. Gehringer

Pipeline scheduling: software pipelining

- Software pipelining: an alternative method to re-organize loops so that ILP is exposed and loop-carried dependences are eliminated.

```
for (1=1, i<100,i++)
      { x:= A[i];
        x:=x+1;
       A[i]:=x}
```

1	2	3	4	**97**	98	99
load A[1]						
incr A[1]	load A[2]					
store A[1]	incr A[2]	load A[3]				
	store A[2]	incr A[3]	load A[4]			
		store A[3]	incr A[4]	•••		
			store A[4]	•••		
				•••		
				•••	•••	
				•••	•••	
				store A[97]	incr A[98]	load A[99]
					store A[98]	incr A[99]
						store A[99]

Pipeline scheduling: software pipelining

- Software pipelining: an alternative method to re-organize loops so that ILP is exposed.

```
for (1=1, i<100,i++)
     { x:= A[i];
        x:=x+1;
       A[i]:=x}
```

1	2	3	4	97	98	99
load A[1]						
incr A[1]	load A[2]					
store A[1]	incr A[2]	load A[3]				
	store A[2]	**incr A[3]**	**load A[4]**			
		store A[3]	incr A[4]	•••		
			store A[4]	•••		
				•••		
				•••	•••	
				•••	•••	
				store A[97]	incr A[98]	load A[99]
					store A[98]	incr A[99]
						store A[99]

Pipeline scheduling: software pipelining

1	2	3	4	97	98	99
load A[1]						
incr A[1]	load A[2]					
store A[1]	incr A[2]	load A[3]				
	store A[2]	incr A[3]	load A[4]			
		store A[3]	incr A[4]	•••		
			store A[4]	•••		
				•••		
				•••	•••	
				•••	•••	
				store A[97]	incr A[98]	load A[99]
					store A[98]	incr A[99]
						store A[99]

```
for (i=1, i<98,i++)
    { A[i]:=y
      y:=x+1;          loop
      x:=A[i+2];
    }
```

Pipeline scheduling: software pipelining

1	2	3	4	97	98	99
load A[1]						
incr A[1]	load A[2]					
store A[1]	incr A[2]	load A[3]				
	store A[2]	**incr A[3]**	**load A[4]**			
		store A[3]	incr A[4]	•••		
			store A[4]	•••		
				•••		
				•••	•••	
				•••	•••	
				store A[97]	incr A[98]	load A[99]
					store A[98]	incr A[99]
						store A[99]

```
x:=A[1];
y:=x+1;                              PREAMBLE
x:=A[2];
for (i=1, i<98,i++)
     { A[i]:=y
       y:=x+1;                loop
       x:=A[i+2];
      }
A[98]:=y;
y:=x+1;                              POSTAMBLE
A[99]=y;
```

Basic compiler techniques

- A compiler embedding advanced analysis techniques is able to produce optimised code, as the example just considered, provided it has info about the target architecture.
- Obviously, a longer compile time is required for the optimization.
- A worse performance can also be expected, if the code is run on machines with the same ISA, but a different architecture (as an instance, different pipeline stages).
- *Question: the optimized code could eventually not work on other architectures (with the same ISA, of course)?*

Basic compiler techniques

- There are actually other problems not discussed so far, that are to be tackled by the compiler:
  - how many rename registers can be used for unrolling ? (namely, how many are available at the ISA level?) When registers run out, the compiler must resort to RAM for temporary storage, which is very inefficient.
  - What about loops whose terminating condition cannot be pre-computed (as in *while-do* or *repeat-until*)?
  - Loop-unrolling produces a longer code than the original. So, there is a chance that Instruction Memory cache misses are more frequent, which makes advantages of unrolling vanish partially (or totally).

Static multiple issue

- If one does not consider these problems, the advantages of compile time ILP are even more evident in multiple issue processors.
- Let us take into consideration the superscalar MIPS architecture already discussed, capable of issuing two instructions per clock cycle:
  - a load / store / branch / integer ALU and
  - any FP operation
- Let us suppose that the previous example has been unrolled over 5 cycles, and let us use instruction latencies discussed in chart n. 7.

Static multiple issue

- Here is the outcome of the execution: it takes 12 clock cycles, that is 2,5 clock cycles per iteration, to be compared with 3,5 in the same processor, with no multiple issue (Hennessy-Patterson, Fig. 4.2):

loop:	int. instruction	FP instruction	clock cycle
	LD F0, 0 (R1)		1
	LD F6, -8 (R1)		2
	LD F10, -16 (R1)	FADD F4, F0, F2	3
	LD F14, -24 (R1)	FADD F8, F6, F2	4
	LD F18, -32 (R1)	FADD F12, F10, F2	5
	SD F4, 0 (R1)	FADD F16, F14, F2	6
	SD F8, -8 (R1)	FADD F20, F18, F2	7
	SD F12, -16 (R1)		8
	DADD R1, R1, #-40		9
	SD F16, 16 (R1)		10
	BNE R1, R2, Loop		11
	SD F20, 8 (R1)		12

Static multiple issue: VLIW

- Superscalar processors using dynamic ILP decide “on the run” on the number of instructions to be issued at each clock cycle. These processors use fairly simple compilers, but require more hardware, to minimize stalls and maximize the number of issued instructions.
- Static multiple issue processors issue a pre-defined number of instructions; it is the task of the compiler to prepare a sequence of instructions to be executed by the CPU in an order that maximizes performances, using the techniques just considered (and more advanced ones).
- Actually, the compiler prepares a series of “bundles” (instruction packets) each containing a fixed number of instructions (a few could be no-ops).

Static multiple issue: VLIW

- Each **issue packet** contains independent instructions, that can be executed in parallel in the pipeline.
- The compiler sets up these packets by reordering instructions (as in the examples just considered) checking and eliminating dependencies, a job that need not be done by the CPU.
- As a consequence, the CPU has a simpler architecture, and the issue phase is shorter, since it is no longer necessary to check instructions at run time to detect dependencies: this analysis has already been carried out by the compiler.

Static multiple issue: VLIW

- This approach is called **VLIW** (**V**ery **L**ong **I**nstruction **W**ord), since the first architectures that deployed it (beginning of 80s), had very long machine instructions (128 bits or more), each specifying multiple independent operations to be executed in parallel by the CPU.

- The name **EPIC** (**E**xplicitly **P**arallel **I**nstruction **C**omputer) used by Intel in the IA-64 architecture denotes the same approach.

Static multiple issue: VLIW

- The larger the number of instructions that can be issued together by a processor, the more advantageous the VLIW approach.
- The overhead incurred upon by a superscalar processor to check at run-time instruction dependencies grows not linearly with the number of instructions (each instruction must be checked against all others that can potentially be issued).

VLIW: an example

- Let us consider a VLIW processor capable of issuing 5 instructions, with the following functional units:
  - integer unit (ALU) that handles branches as well
  - two floating point units
  - two memory access units
- To completely exploit these computing resources, the code must embed enough instruction parallelism, and the compiler must be able to extract it, using loop unrolling and pipeline scheduling.
- Let us re-examine the previous example, and let us assume a level of unrolling of 7 for iterations. The code, arranged in independent packet by the compiler, could arrive to the CPU as follows:

VLIW: an example

Hennessy-Patterson, Fig 4.5:

packet / n. clock	**memory ref. 1**	**memory ref. 2**	**FP op. 1**	**FP op. 2**	**integer / branch op.**
1	LD F0,0(R1)	LD F6,-8(R1)	*no-op*	*no-op*	*no-op*
2	LD F10,-16(R1)	LD F14,-24(R1)	*no-op*	*no-op*	*no-op*
3	LD F18,-32(R1)	LD F22,-40(R1)	FADD F4,F0,F2	FADD F8,F6,F2	*no-op*
4	LD F26,-48(R1)	*no-op*	FADD F12,F10,F2	FADD F16,F14,F2	*no-op*
5	*no-op*	*no-op*	FADD F20,F18,F2	FADD F24,F22,F2	*no-op*
6	SD F4,0(R1)	SD F8,-8(R1)	FADD F28,F26,F2	*no-op*	*no-op*
7	SD F12,-16(R1)	SD F16,-24(R1)	*no-op*	*no-op*	DADD R1,R1,#-56
8	SD F20,24(R1)	SD F24,16(R1)	*no-op*	*no-op*	*no-op*
9	SD F28,8(R1)	*no-op*	*no-op*	*no-op*	BNE R1,R2,Loop

VLIW: an example

- To be noted:
- not all slots within a packet are used, so they are filled in with no-ops. Generally, it is not always possible to use all slots in each packet: code and architecture limit the actual usage. In this example, utilization is roughly 60%.
- 9 clock cycles are required to execute 7 for iterations, an average of 9/7 = 1.29 clock cycles for iteration.
- As already stated, a processor with a different architecture (as an instance, different FUs or different latencies) would yield different performances for the same code.

Advanced techniques

- VLIW leverages on more advanced compilation techniques, to extract ILP at compile time. Here are some:
  - **Static branch prediction**. Assumes that branches exhibit always the same behaviour, or analyses code to infer their behaviour.
  - **Loop Level Parallelism**. Techniques to extract parallelism for iterations, if there are loop-carried dependencies among iterations.
  - **Symbolic loop unrolling**. The loop is not "unrolled", but each packet is filled with independent instructions, even from different iterations.
  - **Global code scheduling**. Tries to collapse code from different basic blocks (basic block = a linear sequence of instructions delimited by conditional instructions, including loops) into sequences of independent instructions.

Hardware support to Static ILP: predicative instructions

- Compiler techniques to extract ILP work properly when branches can be correctly predicted (as in for loops).
- When branch behaviour is dubious, control dependencies limit to a large extent any attempt to exploit parallelism among instructions.
- To partially overcome the problem, a solution is extending the ISA with new **predicative** or **conditional instructions**.

Hardware support to Static ILP: predicative instructions

- These instructions can be used to eliminate some branches, by transforming a control dependency into a data dependency, thus potentially increasing the performances of the processor.

- Actually, predicative instructions are useful also in dynamic ILP, since they allow to cancel some branches.

Predicative or conditional instructions

- Note: in the following we'll use interchangeably *predicative* and *conditional instruction*.
- A predicative instruction contains a condition that is evaluated as part of the execution of the instruction.
  - if the condition is found to be true, the remaining part of the instruction is executed normally.
  - otherwise, the instruction is transformed into a **no-op**.
- Many recent architectures have some form of conditional instruction.

Predicative or conditional instructions

- The simplest predicative instruction is the **conditional move**: it moves the content of a register to another one if the associated condition is true. This instruction allows to avoid branches, in some circumstances.
- Let us consider the C-language instruction **"if (A == 0) S = T;"** and let us assume that R1, R2, R3 hold espectively values A, S, and T. The C-code can be translated into the following MIPS instructions:

```
        BNEZ  R1, Jump        // branch if not zero
        ADD   R2, R3, R0      // R0 always contains 0
Jump:
```

Predicative or conditional instructions

- With a new conditional instruction that is carried out only if the third operand is 0, the IF could be translated into the following instruction:

```
CMOVZ     R2, R3, R1        // mov R3 in R2 if R1 == 0
```

- The *true* control dependency (ADD depends on BNEZ) is transformed into in a *possible* data dependency, according to the instructions that precede the CMOVZ.

- Eliminating these branches has a very positive impact on performances in actual pipelines, where branch execution takes a lot of clock cycles. These branches cannot be easily predicted (as in for loops) and most times they cannot be predicted at all. Any prediction results in a 50% misprediction!

Predicative or conditional instructions

- In multiple issue processors, multiple branches could be scheduled for execution in the same clock cycle, and prediction gets more and more casual, since branches can depend on one another.

  if (A == B) {A = A+1} else if (C == D) {C = C+1}

  in this case, the second if *depends on the first one, and it is quite possible that all instructions get issued in the same clock cycle. How can the second* if *be scheduled, if the first is still being evaluated?*

- Some architectures simply rule out issuing multiple branches in the same clock cycle.
- Conditional move instructions allow to get rid of some branches, thus increasing the CPU issue rate.

Predicative or conditional instructions

- An example: a piece of code (a) with an if-then-else can be translated into a machine code with two branches (b) or into four predicative instructions (c) (Tanenbaum, Fig. 5.52):

```
if (R1 == 0) {
    R2 = R3;
    R4 = R5;
} else {
    R6 = R7;
    R8 = R9;
}
```
(a)

```
    CMP R1,0
    BNE L1
    MOV R2,R3
    MOV R4.R5
    BR L2
L1: MOV R6,R7
    MOV R8,R9
L2:
```
(b)

```
CMOVZ R2,R3,R1
CMOVZ R4,R5,R1
CMOVN R6,R7,R1
CMOVN R8,R9,R1
```
(c)

Predicative or conditional instructions

- Conditional move instructions cannot eliminate branches that control other type of instructions.
- Thus, some architectures implement **full predication**: all instructions can be controlled by a predicate.
- These architectures use specific predicative registers that store the outcome of tests from preceding instructions, so that subsequent instructions can be controlled accordingly (thus becoming predicative themselves).

Predicative or conditional instructions

- The Itanium processor has predicative registers used in couples (P1, P2, ...) and instructions to set such registers.
- As an examples, "CMPEQ R1, R2, P4" sets P4 = 1 and P5 = 0 if R1 == R2. Otherwise, the instruction sets P4 = 0 and P5 = 1.
- Predicative registers are used to control execution of other instructions. Here is an example (Tanenbaum, Fig. 5.53):

```
if (R1 == R2)
    R3 = R4 + R5;
else
    R6 = R4 – R5
```

(a)

```
     CMP R1,R2
     BNE L1
     MOV R3,R4
     ADD R3,R5
     BR L2
L1:  MOV R6,R4
     SUB R6,R5
L2:
```

(b)

```
CMPEQ R1,R2,P4
<P4> ADD R3,R4,R5
<P5> SUB R6,R4,R5
```

(c)

Predicative or conditional instructions

- Predicative instructions are usefull especially in small *if-then-else* constructs, since they eliminate difficult to predict branches; they ease the implementation of more complex statis ILP techniques.
- Predication is however limited by some factors:
  - nullified predicative instructions have used CPU resources, and have prevented/limited the execution od other instructions.
  - Complex brnch combinations cannot be easily transformed into conditional instructions.
  - Conditional instructions can take longer to execute than conventional instructions.

Predicative or conditional instructions

- Many architecture only feature a few simple conditional instructions condizionali, notably conditional move.
- MIPS, SPARC, Alpha, PowerPC, Pentium, dual core and follow-ups support conditional move.
- Intel IA-64 architecture (Itanium) supports full predication

Static ILP vs dynamic ILP

- While initiallly very different, the two approaches to ILP have converged, due to advances in sw and hw technology.
- As a rule of thumb, dynamic ILP seems to fit better general purpose processors, where a complex hardware is viable (with its associated higher power consumption and cost).
- On the contrary, Static ILP is a foundation for embedded processors, where cost and power consuption are of the utmost importance, where applications require a finite set of programs that can be designed and optimized for best parallelism among instructions and hardware utilization.

The Trimedia TM32

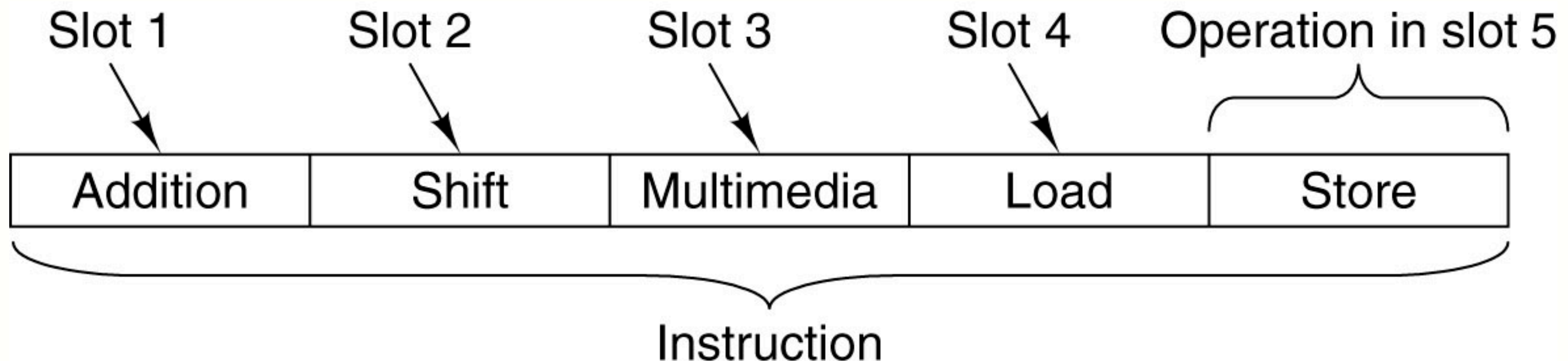
- At the outset of this chapter, it was highlighted that Static ILP suits peculiarly well embedded applications, which cannot rely on a very complex / expensive / energy consuming hardware.

- A typical example of VLIW processor is TriMedia TM32, designed by Philips and used for multimedia applications in CD, DVD, MP3 players, television sets and digital camcorders.

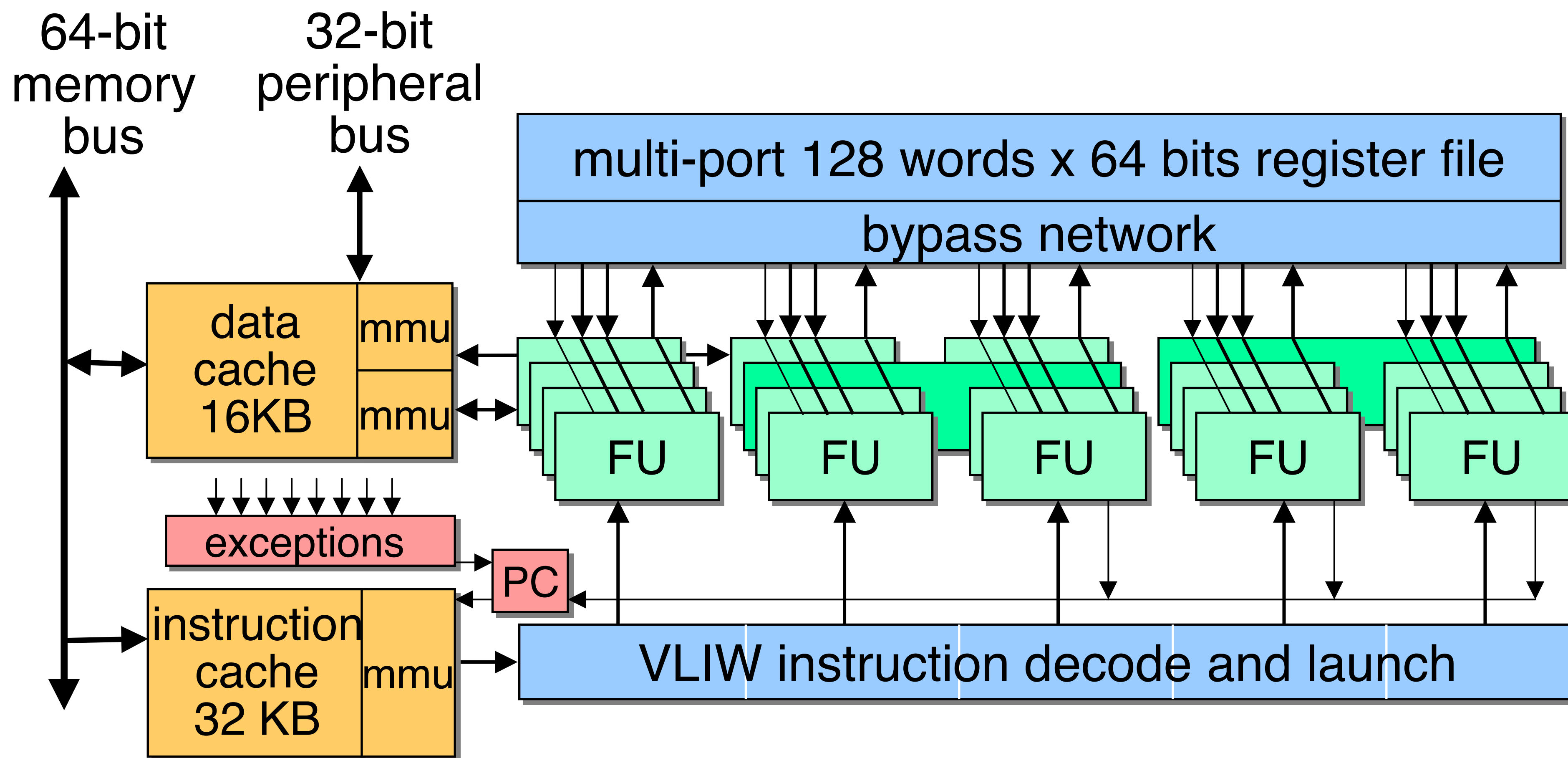
- It is a processor for very specific applications, not for general purpose ones (as is Itanium II).

Trimedia TM32 architecture

- TM32 features clock frequencies between 266 and 300 MHZ; its VLIW instructions specify up to 5 differente operations that are issued in a single clock cycle. Some slot can contain no-ops (Tanenbaum, Fig. 8.3).



Trimedia TM32 architecture



Trimedia TM32 architecture

- TM3260 has 10 families of pipelined FUs, and a not pipelined FP sqrt/div. FU (Tanenbaum, Fig. 8.4):

Unit	Description	#	Lat.	1	2	3	4	5
Constant	Immediate operations	5	1	x	x	x	x	x
Integer ALU	32-Bit arithmetic, Boolean ops	5	1	x	x	x	x	x
Shifter	Multibit shifts	2	1	x	x	x	x	x
Load/Store	Memory operations	2	3				x	x
Int/FP MUL	32-Bit integer and FP multiplies	2	3		x	x		
FP ALU	FP arithmetic	2	3	x			x	
FP compare	FP compares	1	1			x		
FP sqrt/div	FP division and square root	1	17		x			
Branch	Control flow	3	3		x	x	x	
DSP ALU	Dual 16-bit, quad 8-bit multimedia arithmetic	2	3	x		x		x
DSP MUL	Dual 16-bit, quad 8-bit multimedia multiplies	2	3		x	x		

Trimedia TM32 architecture

- TM3260 has 128 general-purpose 32-bit registers, and 4 special registers (PC, PSW, 2 for interrupt handling)
- TM3260 supports *predication* for all instructions, by associating them with a register that nullifies execution, if set to 0.
- TM3260 has no run-time check on compatibility among operations in an instruction, or among consecutive instructions: it simply executes them.
- It is the task of the compiler to schedule operations within instructions , and the sequence of instructions, so that pipeline stalls are minimized and FU utilization is maximized.

Trimedia TM32 architecture

- Hardware complexity is minized, and compilers produce very optimazed code for the specific architecture (they can use a lot of compile time …).
- The single flaw of this approach is code size, that is always much larger than analogous code for RISC processors.
- The code, usually executed from a ROM, is thus compressed: instructions are de-compressed only at fetch time, when they are read from the I-cache.
- Still, code is twice as large as in a similar RISC (next chart: Hennessy-Patterson, Fig. 3.23)

Multi-issue: approaches

name	issue structure	hazard detection	scheduling	characteristic	examples
Superscalar (static)	dynamic	hardware	static	in-order execution	Sun UltraSPARC II/III
Superscalar (dynamic)	dynamic	hardware	dynamic	some out-of-order execution	IBM Power2
Superscalar (speculative)	dynamic	hardware	dynamic with speculation	out-of-order execution with speculation	Intel CPUs, MIPS R10K, Alpha 21264, HP PA 8500, IBM RS64III
VLIW/LIW	static	software	static	no hazards between issue packets	Trimedia, i860
EPIC	mostly static	mostly software	mostly static	explicit dependencies marked by compiler	Itanium, Itanium II