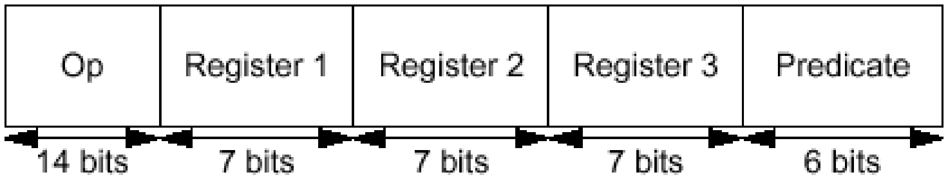
1 IA64

IA64 steht für “Intel Architecture”. Sie ist der Nachfolger der IA32, die in den Prozessoren der Pentium Serie und ihren Clones (z.B. AMD Athlon) implementiert ist.
IA64 ist eine gemeinschaftliche Entwicklung von Intel und HP, sie wird seit 1994 entwickelt, und wahrscheinlich noch dieses Jahr im Intel “Itanium” zum Einsatz kommen. Ausserdem wird sie auf Unix Rechnern (HP, SGI) verwendet werden.
Sie ist natürlich eine 64 Bit Architektur, arbeitet daher mit 64 Bit Integerzahlen und kann 2^{64} Bytes Speicher adressieren.
Entscheidend ist aber, wie die IA64 Parallelausführung von Instruktionen fördert.

1.1 Codierung der Instruktionen

Jede Instruktion verweist über eine Nummer auf ein Prädikatsregister. Ein solches Register enthält einen boolschen Wert. Ist der Wert "false", wird die Instruktion nicht ausgeführt.



Wie man sieht, erlaubt IA64 $2^{14} = 16394$ verschiedene Befehle, mit jeweils bis zu drei Registern als Argumente, zum Beispiel:

```
add r1 = r2, r3
```

1.1.1 Bundles

Es werden immer drei Instruktionen zu einem “Bundle” zusammengefasst. In jedem Bundle ist ein “Template” enthalten, das angibt, welche Recheneinheiten die Instruktionen jeweils benötigen.

Insgesamt ist jedes Bundle genau 128 Bits lang.

1.1.2 Bundles in IA64 Assembler

```
{ .mfi
ld4 r14 = [r56]
fadd f10 = f12, f13
add r16 = r18, r19
}
```

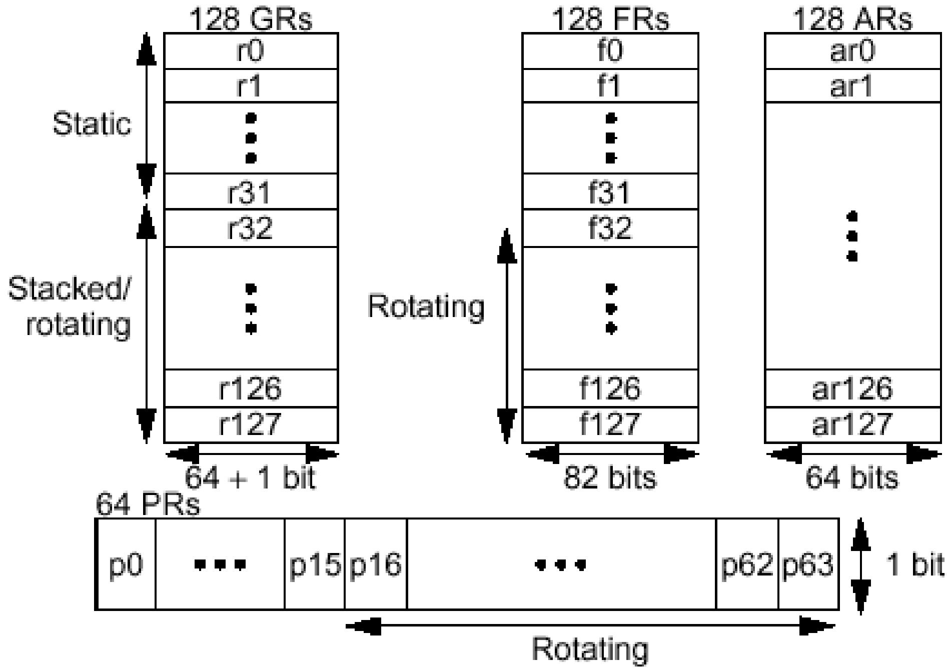
“.mfi” ist hier das Template. Im Normalfall werden der Templatestring und die geschweiften Klammern weggelassen, der Assembler erzeugt sie selbständig.

1.1.3 Instruktionsblöcke

Voneinander abhängige Instruktionen werden durch “;;” getrennt. So einstehen Instruktionsblöcke.
Theoretisch könnten Instruktionsblöcke also in einem Schritt parallel ausgeführt werden. Der Itanium ist in der Lage, bis zu zwei Bundles auszuführen, falls genügend Recheneinheiten vorhanden sind, um die Instruktionen auf einmal zu starten.
Auf diesem Weg kann die CPU direkt erkennen, was parallel ausgeführt werden kann. Beim IA32 ist das anders. Dort stehen die Instruktionen sequenziell nacheinander - die CPU muss “raten”, welche Instruktionen parallel ausgeführt werden können.
Das Konzept des IA64 heisst **Parallele Semantik**, das Konzept des IA32 **Sequenzielle Semantik**.

1.2 Register

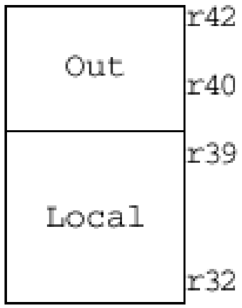
1.2.1 Register Files



IA64 besitzt “General”, “Floating Point”, “Application” und “Predicate” Register.

1.2.2 Register Rotation

Wie auch die SPARC, hat IA64 Registerfenster. Allerdings kann bei IA64 die Anzahl der lokalen und der Ausgaberegister eingestellt werden. Ausgaberegister werden dann nach einem “call” zu Eingaberegistern, die lokalen verschwinden durch die Rotation.
“save” und “restore” existieren, im Unterschied zur SPARC, nicht.



2 EPIC

EPIC steht für “Explicit Parallel Instruction Computing’. Entscheidend ist das Wort “explicit”. Denn parallel arbeiten auch schon die IA32 Rechner, nur ermöglichte IA32 es nicht, Parallelität explizit auszudrücken.
IA64 verwendet folgende Ansätze, um besser parallel zu arbeiten:

- Predication
- Control Speculation
- Data Speculation
- Parallel Branches
- Software Pipelining

2.1 Beispiel: Vergleiche

Zusammengesetzte Vergleiche können in IA64 parallel formuliert werden. Betrachte den folgenden C Code:

```
if ((a==0) || (b<=5) || (c!=d) || (f & 0x2)) {
    r3 = 8;
}
```

Wie wird dieser Code auf IA32 kodiert, wie auf IA64?

2.1.1 Vergleich in IA32 Assembler

```
cmp   0,a
je    eins
cmp   5,b
jle   eins
cmp   c,d
jne   eins
and   2,f
test  f,f
jne   eins
jmp   zwei
eins:
mov   8,r3
zwei:
```

Solche Vergleiche werden also als Folge von “cmp” Instruktionen kodiert.

2.1.2 Vergleich in IA64 Assembler

```
cmp.ne p1 = r0, r0
add  t = -5, b;;

cmp.eq.or p1 = 0, t
cmp.ge.or p1 = 0, t
cmp.ne.or p1 = c, d
tbit.or p1 = 1, f, 1;;

(p1) mov r3 = 8
```

Zuerst wird das Prädikat p1 auf “false” initialisiert.
Der Vergleich wird in einem einzigen Block kodiert. Der zweite Instruktionsblock enthält den gesamten Vergleich, dessen Resultat in p1 gesichert wird.
Danach wird “mov” bedingt ausgeführt.

2.2 Bedingte Ausführung durch Prädikate

Prädikate ersetzen also Branches bei bedingtem Code. Das ist besonders deshalb wichtig, weil Branch Prediction bei bedingtem Code oft nicht effektiv ist.
Da es 64 Prädikatsregister gibt, sind verschachtelte "if" Statements bis zur Tiefe 6 ohne Branch möglich. Diese Grenze wird selten erreicht, noch seltener sinnvoll.
Prädikate kosten keine zusätzliche Ausführungszeit, da sie für jede Instruktion implizit benutzt werden.

2.3 Control Speculation

IA64 ist in der Lage, bestimmte Instruktionen - z.B. das Laden eines Wertes aus dem Hauptspeicher - über Branches hinweg nach vorne zu ziehen. Am ursprünglichen Ausführungsort des Befehls wird ein “chk.s” eingefügt, das dann prüft, ob durch das Vorziehen eine Exception ausgelöst wurde. Das kann nur passieren, falls der Code nicht direkt nach der vorgezogenen Ausführung angesprungen wurde.
Falls eine Exception ausgelöst wurde, wird ein Korrekturcode gestartet.
Control Speculation spekuliert also auf einen bestimmten Verlauf des Kontrollflusses.

2.4 Data Speculation

Viele Programmiersprachen, beispielsweise C, haben Pointer-Datentypen. Es ist nicht möglich zu erkennen, ob ein Load durch einen Pointer denselben Speicherbereich benutzt, wie ein Store durch einen zweiten Pointer, ohne die Pointer, also Werte im Speicher, zu vergleichen. Da man gerade die Verzögerung des Ladens vermeiden will, kann das nicht die Lösung sein.
Daher verwendet man Data Speculation. Man zieht das Laden eines Wertes vor das Schreiben, und spekuliert, daß das Schreiben den zu ladenen Wert nicht verändert hätte.

2.4.1 Realisierung der Data Speculation - ALAT

Lädt man einen Wert spekulativ, wird seine Adresse in die “Advanced Load Address Table” (ALAT) eingetragen.
Ob die Spekulation falsch war, wird über die ALAT ermittelt. Wenn die Hardware schreibt, wird die beschriebene Speicheraddresse in der ALAT gesucht, und - falls vorhanden - gelöscht.
An der Stelle, an der ursprünglich geladen werden sollte, wird ein “ld.c” eingefügt, das in der ALAT nach der Ladeaddresse sucht. Falls nicht vorhanden, wird der Wert an dieser Stelle neu geladen.

Wird spekulativ mit dem geladenen Wert gerechnet, z.B. um die Recheneinheiten gleichmässiger auszulasten, verwendet man “chk.a” zum Überprüfen der Spekulation. “chk.a” funktioniert genau wie “ld.c”, nur dass statt des neu Ladens beliebiger Korrekturcode gestartet werden kann.

2.5 Branches

Innerhalb einer Instruction Group können mehrere Branch Instruktionen stehen. Hier gibt es eine Ausführungsreihenfolge, und zwar von oben nach unten. Man kann Branches also “parallel” ausführen - die Branches werden zeitgleich vorbereitet, und von allen Branches, deren Prädikat “true” ist, wird der oberste tatsächlich ausgeführt.
So hat man eine effiziente Methode, “switch”, oder “else if” Statements zu realisieren.

2.6 Zählende Schleifen

Oft kommen in Programmen kurze “for” Schleifen vor, die z.B. einen Wert über eine bestimmte Anzahl Iterationen bestimmen. Branch Prediction schlägt bei diesen Schleifen immer am letzten Branch fehl.

Daher führt man “LC” (Loop Counter) und “CLOOP” (Close Loop) ein.

LC ist ein Integer Register für die Anzahl der Durchläufe, CLOOP branch't normal, und verkleinert LC dabei um 1. Falls LC 0 ist, branch't CLOOP nicht mehr.

```
mov lc = 10
.label loop
ld8 rt = [ra];;
st8 [rb] = rt
br.cloop #loop
```

2.7 Software Pipelining

Bei IA64 wird “Modulo Scheduling” angewandt.
Die Idee ist, mehrere Iterationen einer Schleife übereinanderzulegen, um nicht auf die Beendigung eines Durchlaufs warten zu müssen.
Dazu gibt es ein weiteres Register: “EC” (Epilogue Count), das angibt, wieviele Durchläufe noch beendet werden müssen, nachdem der Loop Count 0 ist.
Ausserdem eine neue Instruktion: “ctop”. Sie senkt LC bei jedem Branch. Falls LC 0 ist, wird EC gesenkt. Falls EC auch 0 ist, ist die Schleife beendet. An ctop sind Prädikatsoperationen gebunden: p16 wird “true”, falls $LC > 0$, sonst “false”, und die restlichen Prädikatsregister ($> p16$) werden um 1 nach rechts verschoben.

2.7.1 Beispiel

Software Pipeline			
	Predicate		
Cycle	p16	p17	
1	ld		
2		st	br.ctop

Ein Schleifendurchlauf. Es wird hier angenommen, ld bräuchte einen Takt zum Laden eines Wertes.
Man initialisiert nun p16 auf “true”, und p17 auf “false”. Damit wird am Schleifenanfang das Speichern erst nach dem Branch ausgeführt, also nachdem das Laden erfolgt ist.

Durch den ctop Mechanismus verläuft die Ausführung dann so:

Predicated Modulo-Scheduled Loop Execution Sequence

Cycle	Stage Predicate p16	Stage Predicate p17		Predicate Value p16	Predicate Value p17	LC	EC
1	ld1		br.ctop	1	0	n-1	2
2	ld2	st1	br.ctop	1	1	n-2	2
3	ld3	st2	br.ctop	1	1	n-3	2
n	ldn	stn-1	br.ctop	1	1	0	2
n+1		stn	br.ctop	0	1	0	1
after loop exit				0	0	0	0

2.7.2 Performancegewinn bei High-Latency Instruktionen

Nun wird angenommen, ld bräuchte 4 Takte. Das ist deutlich realistischer.
Also stallt das Speichern normalerweise um 3 Takte.

Software Pipeline			
	Predicate		
Cycle	p16	p20	
1	ld		
2			
3			
4			
5		st	br.ctop

Durch Modulo Scheduling wird diese Latenzzeit umgangen.

Die Ausführung sieht dann so aus:

Predicated Modulo-Scheduled Loop Execution Sequence

	Stage Predicate			Predicate Value					LC	EC
Cycle	p16	p20		p16	p17	p18	p19	p20		
1	ld1		br.ctop	1	0	0	0	0	n-1	5
2	ld2		br.ctop	1	1	0	0	0	n-2	5
3	ld3		br.ctop	1	1	1	0	0	n-3	5
4	ld4		br.ctop	1	1	1	1	0	n-4	5
5	ld5	st1	br.ctop	1	1	1	1	1	n-5	5
n-1	ldn-1	stn-5	br.ctop	1	1	1	1	1	1	5
n	ldn	stn-4	br.ctop	1	1	1	1	1	0	5
n+1		stn-3	br.ctop	0	1	1	1	1	0	4
n+2		stn-2	br.ctop	0	0	1	1	1	0	3
n+3		stn-1	br.ctop	0	0	0	1	1	0	2
n+4		stn	br.ctop	0	0	0	0	1	0	1
after				0	0	0	0	0	0	0