

# **Lista Rozkazów:**

---

**Język komputera (cd.)**

# Procedury

```
int funkcja (int n){  
    int i,j;  
    (...)  
    return j;  
}
```

```
main () {  
    int i,j;  
    i=funkcja(i);  
    ...  
    j=funkcja(i);  
}
```

- funkcja operuje na pewnych rejestrach, być może na tych samych co program główny
- po skoku do funkcji musimy wrócić do miejsca z którego skakaliśmy -- funkcja jest wywoływana z różnych miejsc; musimy zapamiętać adres powrotu
- do funkcji w jakiś sposób musimy przekazać parametry i w jakiś sposób musimy otrzymać wyniki.

# Procedury

- Wymagane kroki
  1. Przekaż parametry do procedury
  2. Przekaż sterowanie do procedury
  3. Przygotuj miejsce na dane procedury
  4. Wykonaj procedurę
  5. Przekaż wyniki stronie wywołującej
  6. Powrót do miejsca wywołania

# Użycie rejestrów

- \$a0 – \$a3: parametry wywołania (rej. 4 – 7)
- \$v0, \$v1: zwracane wyniki (rej. 2 i 3)
- \$t0 – \$t9: zmienne tymczasowe, mogą być zmienione przez procedurę
- \$s0 – \$s7: muszą być odtworzone po zakończeniu procedury
- \$gp: global pointer (rej. 28)
- \$sp: stack pointer (rej. 29)
- \$fp: frame pointer (rej. 30)
- \$ra: return address (rej. 31)

# Wywołanie procedury

- Wywołanie: jump and link  
`jal ProcedureLabel`
  - Adres kolejnego rozkazu umieszczany w `$ra`
  - Skok pod wskazaną etykietę
- Powrót z procedury: jump register  
`jr $ra`
  - Kopiuje `$ra` do licznika rozkazów PC
  - Może być użyty do innych celów
    - Np. konstrukcje case/switch

# Procedura „liść”

- Procedura nie wywołująca innych procedur

- Kod C:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

- Parametry g, ..., j w \$a0, ..., \$a3
- f w \$s0 (musimy zachować \$s0 na stosie)
- Wynik w \$v0

# Procedura „liść”

- Kod MIPS:

leaf_example:			
addi	\$sp,	\$sp,	-4
sw	\$s0,	0(\$sp)	
add	\$t0,	\$a0,	\$a1
add	\$t1,	\$a2,	\$a3
sub	\$s0,	\$t0,	\$t1
add	\$v0,	\$s0,	\$zero
lw	\$s0,	0(\$sp)	
addi	\$sp,	\$sp,	4
jr	\$ra		

odłóż \$s0 na stosie

kod procedury

Wynik

odtworzenie \$s0

Powrót

# Procedury wywołujące inne

- W szczególności procedury rekurencyjne
- Strona wywołująca musi odłożyć na stosie:
  - Swój adres powrotu
  - Parametry i zmienne tymczasowe, które będą potrzebne po powrocie
- Po powrocie z wywoływanej procedury odtwarza powyższe



# Przykład procedury rekurencji.

- Kod C:

```
int fact (int n)
{
    if (n < 1) return f;
    else return n * fact(n - 1);
}
```

- Parametr n w \$a0
- Wynik w \$v0

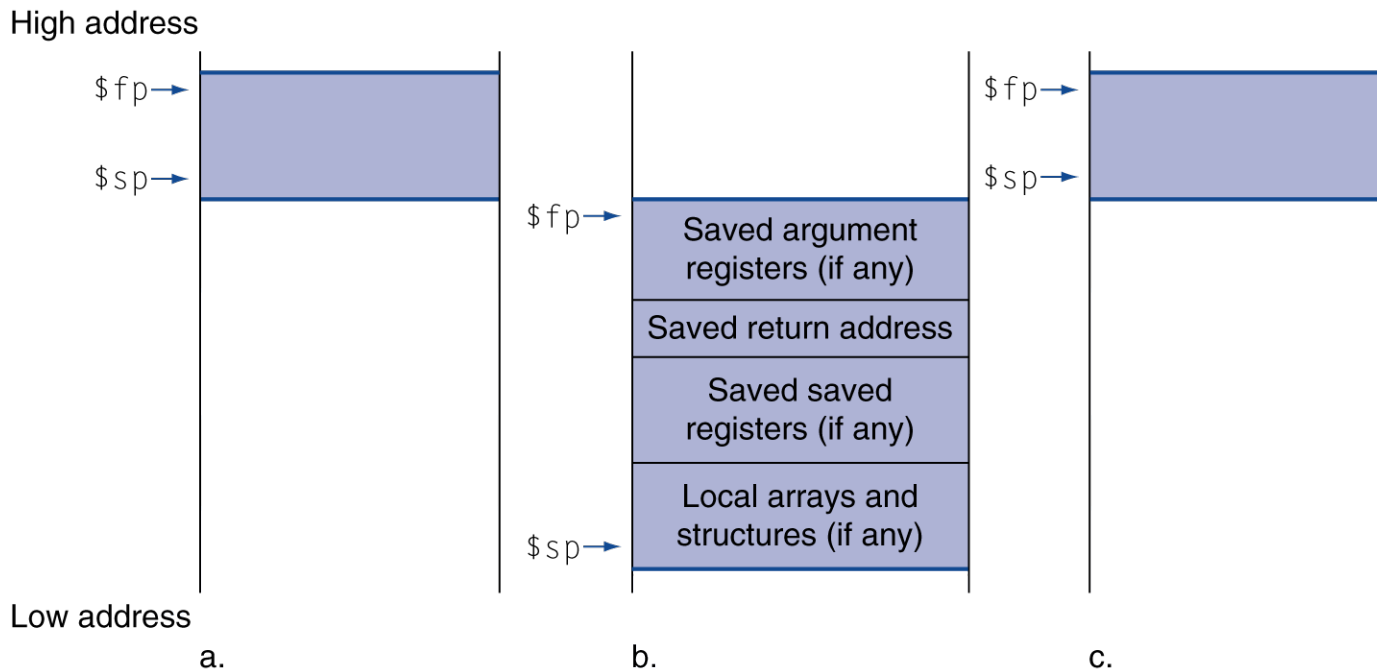
# Przykład procedury rekuren.

## ■ Kod MIPS:

fact:

```
    addi $sp, $sp, -8      # zrób 2 miejsca na stosie
    sw    $ra, 4($sp)      # zachowaj adres powrotu
    sw    $a0, 0($sp)      # zachowaj parametr
    slti  $t0, $a0, 1      # czy n < 1
    beq   $t0, $zero, L1   # jeśli tak, wynik = 1
    addi  $v0, $zero, 1     # usuń 2 elementy ze stosu
    addi  $sp, $sp, 8       # i wróć
L1:  addi  $a0, $a0, -1     # wpp. zmniejsz n
    jal   fact             # wywołanie rekurencyjne
    lw    $a0, 0($sp)      # odtwórz oryginalne n
    lw    $ra, 4($sp)      # i adres powrotu
    addi  $sp, $sp, 8       # usuń 2 elementy ze stosu
    mul   $v0, $a0, $v0     # wykonaj domnożenie do wyniku
    jr    $ra              # i wróć
```

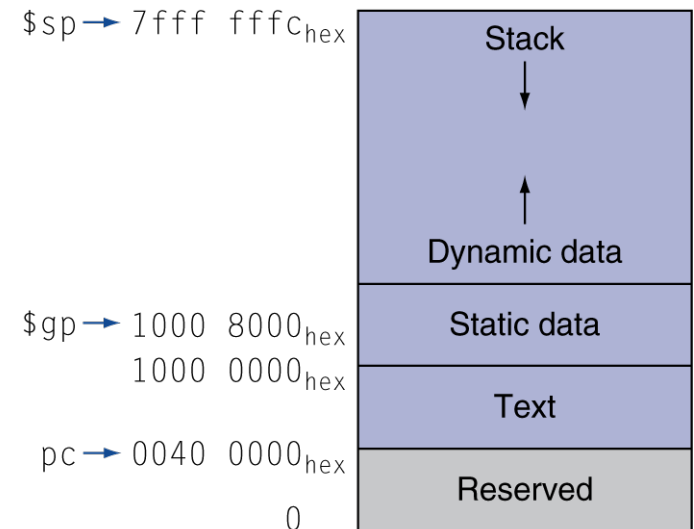
# Dane lokalne na stosie



- Dane lokalne procedury
- Ramka procedury (rekord aktywacji)
- Niekiedy, oprócz \$sp, wygodnie użyć \$fp, ale nie jest to niezbędne

# Układ pamięci

- Text: kod programu
- Static data: zmienne globalne
  - Np. zmienne globalne w C, tablice, napisy...
  - \$gp zainicjowany tak, aby łatwo było się odwoływać do danych za pomocą ujemnego/dodatniego przesunięcia
- Dynamic data: sarta
  - Np. malloc w C, new w Javie
- Stack: zmienne lokalne



# Dane tekstowe

- Znaki kodowane bajtowo
  - ASCII: 128 znaków
    - 95 graficzne, 33 sterujące
  - Latin-1: 256 znaków
    - ASCII, +96 znaków graficznych
- Unicode: kod 32-bit
  - Używany m.in. w Javie,
  - Większość światowych alfabetów, plus inne symbole
  - UTF-8, UTF-16: kodowania zmiennej długości

# Operacje bajtowe/półsłownowe

- Można użyć operacji na bitach
- Operacje na bajtach i połowach słów:

- Przetwarzanie tekstów

lb rt, offset(rs)                      lh rt, offset(rs)

- Roszerzenie arytm. do 32 bitów rt

lbu rt, offset(rs)                      lhu rt, offset(rs)

- Uzupełnianie zerami rt

sb rt, offset(rs)                      sh rt, offset(rs)

- Wysyła mniej znaczącą część rejestru

# Przykład: kopiowanie napisu

- Kod C:

- Zakładamy, że łańcuch zakończony zerem

```
void strcpy (char x[], char y[])  
{ int i;  
  i = 0;  
  while ((x[i]=y[i])!='\0')  
    i += 1;  
}
```

- Adresy x, y w \$a0, \$a1
- i w \$s0

# Przykład: kopiowanie napisu

## ■ Kod MIPS:

strcpy:		
addi	\$sp, \$sp, -4	# 1 miejsce na stosie
sw	\$s0, 0(\$sp)	# zachowaj \$s0
add	\$s0, \$zero, \$zero	# i = 0
L1:	add \$t1, \$s0, \$a1	# adres y[i] w \$t1
lbu	\$t2, 0(\$t1)	# \$t2 = y[i]
add	\$t3, \$s0, \$a0	# adres x[i] w \$t3
sb	\$t2, 0(\$t3)	# x[i] = y[i]
beq	\$t2, \$zero, L2	# kończ pętlę gdy y[i] == 0
addi	\$s0, \$s0, 1	# i = i + 1
j	L1	# na początek pętli
L2:	lw \$s0, 0(\$sp)	# odtwórz \$s0
addi	\$sp, \$sp, 4	# zwolnij miejsce na stosie
jr	\$ra	# powrót



# Przykład: sortowanie

- Implementacja sortowania bąbelkowego
- Kod swap w C (procedura „liść”):

```
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

- Adres v: w \$a0, k: w \$a1, temp: w \$t0

# Procedura swap

swap: sll \$t1, \$a1, 2	# \$t1 = k * 4
add \$t1, \$a0, \$t1	# \$t1 = v+(k*4)
	# (adres v[k])
lw \$t0, 0(\$t1)	# \$t0 (temp) = v[k]
lw \$t2, 4(\$t1)	# \$t2 = v[k+1]
sw \$t2, 0(\$t1)	# v[k] = \$t2 (v[k+1])
sw \$t0, 4(\$t1)	# v[k+1] = \$t0 (temp)
jr \$ra	# wyjście z procedury

# Procedura sortująca w C:

- Wywołuje swap:

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1;
            j >= 0 && v[j] > v[j + 1];
            j -= 1) {
            swap(v, j);
        }
    }
}
```

- Adres v w \$a0, k w \$a1, i w \$s0, j w \$s1

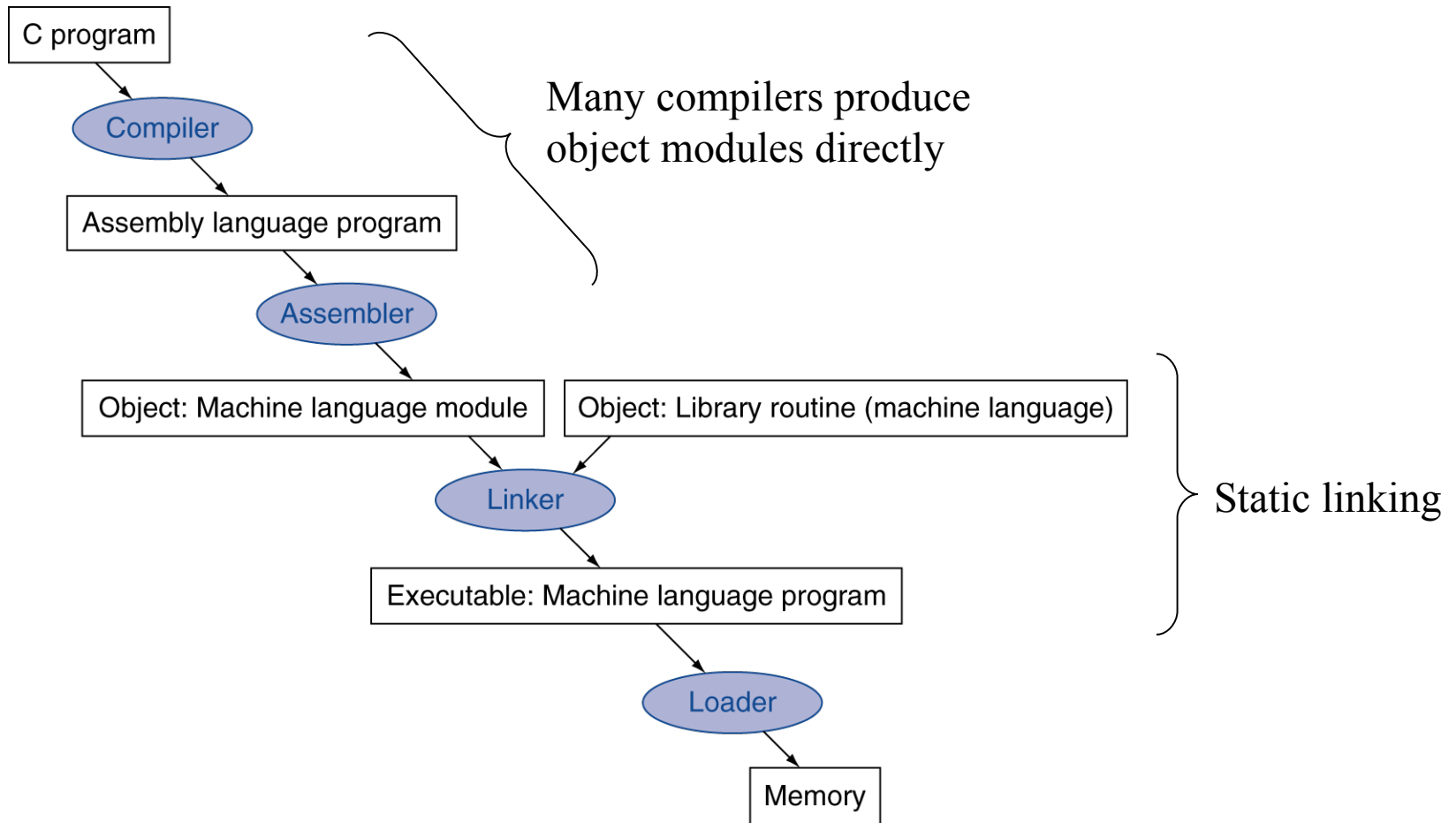
# Procedura sortująca (ciało):

move \$s2, \$a0	# save \$a0 into \$s2	Zapisz param.
move \$s3, \$a1	# save \$a1 into \$s3	
move \$s0, \$zero	# i = 0	
for1tst: slt \$t0, \$s0, \$s3	# \$t0 = 0 if \$s0 ≥ \$s3 (i ≥ n)	Pętla zew.
beq \$t0, \$zero, exit1	# go to exit1 if \$s0 ≥ \$s3 (i ≥ n)	
addi \$s1, \$s0, -1	# j = i - 1	
for2tst: slti \$t0, \$s1, 0	# \$t0 = 1 if \$s1 < 0 (j < 0)	Pętla wew.
bne \$t0, \$zero, exit2	# go to exit2 if \$s1 < 0 (j < 0)	
sll \$t1, \$s1, 2	# \$t1 = j * 4	
add \$t2, \$s2, \$t1	# \$t2 = v + (j * 4)	
lw \$t3, 0(\$t2)	# \$t3 = v[j]	
lw \$t4, 4(\$t2)	# \$t4 = v[j + 1]	
slt \$t0, \$t4, \$t3	# \$t0 = 0 if \$t4 ≥ \$t3	
beq \$t0, \$zero, exit2	# go to exit2 if \$t4 ≥ \$t3	
move \$a0, \$s2	# 1st param of swap is v (old \$a0)	Wywoł. swap
move \$a1, \$s1	# 2nd param of swap is j	
jal swap	# call swap procedure	
addi \$s1, \$s1, -1	# j -= 1	Pętla wew.
j for2tst	# jump to test of inner loop	
exit2: addi \$s0, \$s0, 1	# i += 1	Pętla zew.
j for1tst	# jump to test of outer loop	

# Pełna procedura sortująca

sort:	addi \$sp,\$sp, -20	# make room on stack for 5 registers
	sw \$ra, 16(\$sp)	# save \$ra on stack
	sw \$s3,12(\$sp)	# save \$s3 on stack
	sw \$s2, 8(\$sp)	# save \$s2 on stack
	sw \$s1, 4(\$sp)	# save \$s1 on stack
	sw \$s0, 0(\$sp)	# save \$s0 on stack
	...	# ciało procedury
	...	
exit1:	lw \$s0, 0(\$sp)	# restore \$s0 from stack
	lw \$s1, 4(\$sp)	# restore \$s1 from stack
	lw \$s2, 8(\$sp)	# restore \$s2 from stack
	lw \$s3,12(\$sp)	# restore \$s3 from stack
	lw \$ra,16(\$sp)	# restore \$ra from stack
	addi \$sp,\$sp, 20	# restore stack pointer
	jr \$ra	# return to calling routine

# Tłumaczenie i Uruchamianie



# Pseudorozkazy

- Większość rozkazów asemblera jest reprezentowana w kodzie maszynowym 1-1
- Pseudorozkazy: tłumaczone przez program asemblerujący na rozkaz lub sekwencję „prawdziwych rozkazów”

`move $t0, $t1` → `add $t0, $zero, $t1`

`blt $t0, $t1, L` → `slt $at, $t0, $t1`  
`bne $at, $zero, L`

- `$at` (register 1): rejestr pomocniczy

# Rozkazy zmiennopozycyjne

- Osobne układy procesora i rozkazy
- 32 rejestry zmiennopozycyjne (32-bitowe)
  - \$f0-\$f31
  - Przechowują liczby single precision
  - Pary rejestrów \$f0-\$f1, \$f2-\$f3, itd. przechowują liczby double precision
- Rozkazy: add.s, mul.s, add.d, mul.d, ...
  - Format R; .s – single, .d - double
  - W przypadku rozkazów .d podajemy numery pierwszych rejestrów z pary



# Rozkazy zmiennopozycyjne

- Osobne rozkazy przesyłania danych...
  - lwc1 \$f1, 40(\$s1) , swc1 \$f2, 12(\$s0)

- ... i skoki warunkowe

- c.eq.s \$f1, \$f4  
bc1t Label  
bc1f Label2

- Przykład:

.data

floats: .float -0.123, 3.14

(...)

lwc1 \$f0, floats

lwc1 \$f1, floats + 4

# Wejście/wyjście

- MIPS używa IO „mapowanego w pamięci”
  - Pewne adresy oznaczają urządzenia IO
  - Operacje IO realizowane za pomocą lw, sw
- W SPIM-ie to rozwiązanie nie jest domyślne (ale można je wymusić)
- Urządzenie wejściowe (klawiatura), wyjściowe (monitor) (monitor). Każde ma dwa rejestry:
  - 0xffff0000 - rejestr sterujący wejściem
  - 0xffff0004 - rejestr danych wejściowych (1 bajt)
  - 0xffff0008 - rejestr sterujący wyjściem
  - 0xffff000c - rejestr danych wyjściowych

# Funkcje systemowe

- SPIM oferuje pewną liczbę „funkcji systemowych” realizujących IO
  1. Print Integer (\$a0 = wartość do wydrukowania)
  2. Print Float
  3. Print Double
  4. Print String (\$a0 = adres napisu)
  5. Read Integer (wczytany wynik zwracany w \$v0)
  6. Read Float (wynik w \$f0)
  7. Read Double
  8. Read a String (\$a0 = adres pod jaki należy wpisać, \$a1 = długość przydzielonej pamięci)... I jeszcze parę innych

# Funkcje systemowe

```
.data
str:
.asciiz "the answer ="
.text
li $v0, 4      # print string
la $a0, str
syscall
li $v0, 1      # print integer
li $a0, 5
syscall
```

# ARM & MIPS

- ARM (Advanced RISC Machine)
- Najpopularniejsza obecnie architektura dla urządzeń wbudowanych
- Typu load/store; trzyrejestrowe rozkazy arytmetyczne

	ARM	MIPS
Powstanie	1985r.	1985r.
Długość rozkazu	32 bity	32 bity
Przestrzeń adresowa	32-bitowa	32-bitowa
Wyrównywanie danych	tak	tak
Tryby adres. danych	9	3
Rejestry ogół. przeznacz.	15 × 32-bity	31 × 32-bity
Wejście/wyjście	Memory mapped	Memory mapped

# ARM & MIPS

- ARM ma kilka rozwiązań niespotykanych w innych procesorach RISC. Zwrócimy uwagę na dwa:
  - Wiele (skomplikowanych) trybów adresowania
  - Warunkowe wykonywanie (dowolnych) rozkazów

Co ciekawe, w nowej 64-bitowej wersji ARM 8 w zasadzie zrezygnowano z tych rozwiązań...



# Tryby adresowania

---

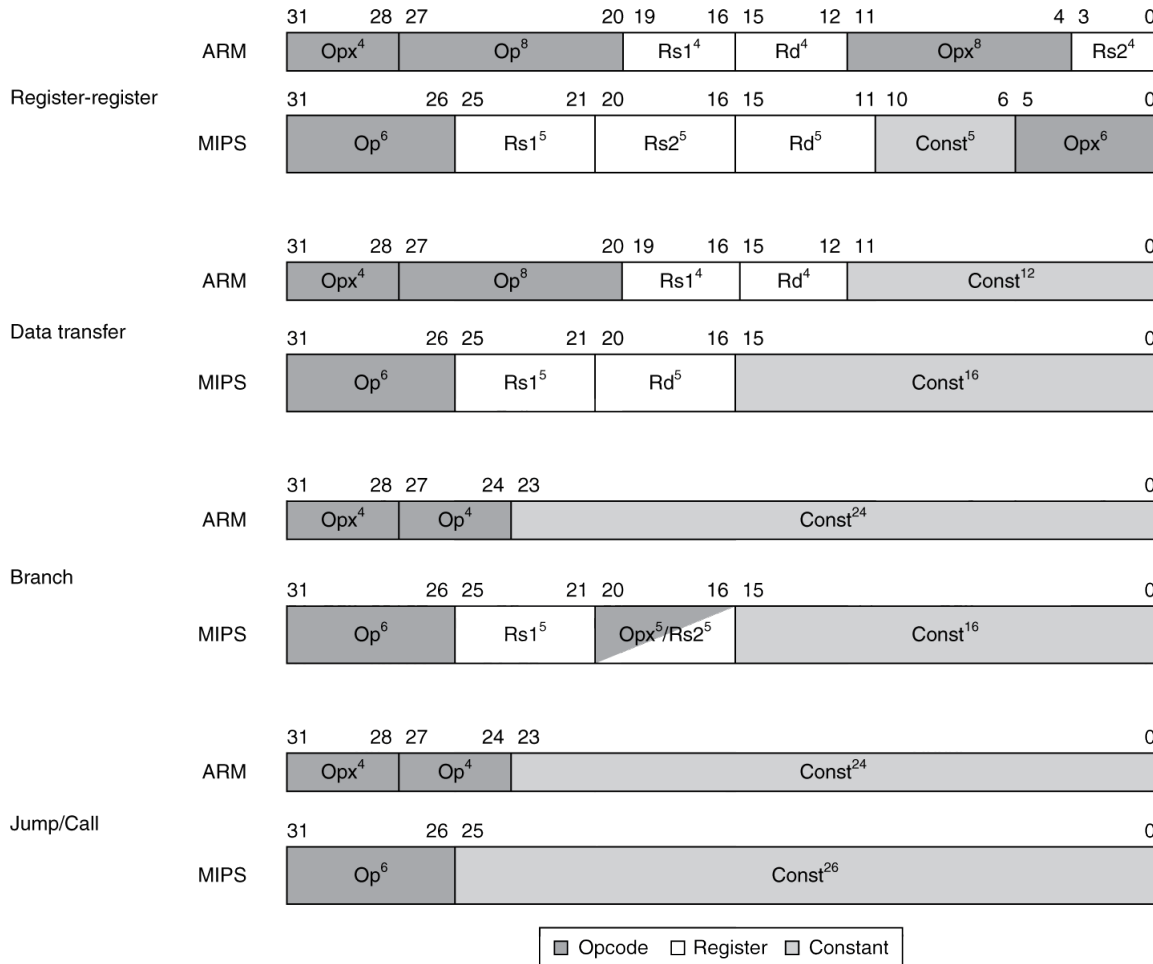
- Rejestrowe
- Natychmiastowe
- Rejestr + przesunięcie
- Rejestr + (przeskalowany) rejestr
- Rejestr + przesunięcie (ze zmianą rejestru)
- Rejestr + rejestr (ze zmianą rejestru)
- Automatyczna inkrementacja/dekrement.
- Względem PC

# Warunkowe wykonywanie

- Specjalny rejestr, z czterema bitami:
  - Z(Zero), N(Negative), C(Carry), V(Overflow)
- Cztery bity każdego rozkazu określają warunek, pod jakim ten rozkaz ma się wykonać
- Ustawiane przez rozkazy porównań...
  - CMP r1, r2 (r1-r2, wynik ustawia ZNCV),
  - CMN (r1+r2), TST (r1 AND r2) , TEQ (XOR)
- ...i rozkazy arytmetyczne: ADDS r1, r2, r3
- Warunkowe wykonanie:
  - ADDEQ r0, r1, r2 – dodaj jeśli Z=0
  - BGE label – skocz jeśli N=V=1 lub N=V=0



# ARM – kodowanie rozkazów



# Przykład

```
while(i != j) {  
    if (i > j)  
        i -= j;  
    else  
        j -= i;  
}
```

```
loop CMP Ri, Rj ; set condition "NE" if (i != j),  
        ; "GT" if (i > j),  
        ; or "LT" if (i < j)  
SUBGT Ri, Ri, Rj ; if "GT" (greater than),  
SUBLT Rj, Rj, Ri ; if "LT" (less than),  
BNE loop ; if "NE" (not equal), then loop
```