Optymalizacja kodu – projekt ‘multacc’

Pliki znajdują się w katalogu pipeline.
- wczytać konfigurację: escape.ecf
- uruchomić architekturę potokowę
- wczytać projekt: multacc.ppr

- ustawić MemAccTime =1
 (Options → Memory Access Time → 1)

- nie używać opóźnionego skoku
 (Options → Delayed
Branching → No…

- wyłaczyć Forwarding
 (Options → Enable Forwarding → Off

- ustawić breakpoint PC = 0x30

- zmierzyć czas działania:
with forwarding = 163
witout forw. = 244

- aktywować i obejrzeć diagramy kolejki

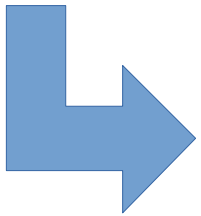
```
0000: 3401003C |            | ADDI R0, 0x003C, R1
0004: 34020000 |            | ADDI R0, 0x0000, R2
0008: 38210004 | loop       | SUBI R1, 0x0004, R1
000C: 04230004 |            | LD   R3, 0x0004(R1)
0010: 04240044 |            | LD   R4, 0x0044(R1)
0014: 14641800 |            | MUL  R3, R4, R3
0018: 08230084 |            | ST   R3, 0x0084(R1)
001C: 0C431000 |            | ADD  R2, R3, R2
0020: 6C01FFE4 |            | BRGE R1, loop
0024: 00000000 |            | NOP
0028: 00000000 |            | NOP
002C: 00000000 |            | NOP
0030: 6000FFFC | halt       | BRZ  R0, halt
```

Optymalizacja kodu – projekt ‘multacc’

Program ‘multacc.cod ’wykonuje 16 cykli pętli, mnożąc liczby z dwóch tablic (int 32-bit).

Poniżej pokazano jego funkcjonalny odpowiednik w języku C:
- wartości tablic A i B znajdują się w pamięci danych pod adresami: 0x00.. 0x3C i 0x40.. 0x7C
- obszar tablicy C (pod adresem 0x80.. 0xBC) jest początkowo wypełniony zerami
- zmienna sterująca pętli jest w rejestrze R1
- zmienna s (akumulator) jest w rejestrze R2 (po zakończeniu programu wynosi 0x0E58)

```
        ADDI R0, 0x003C, R1
        ADDI R0, 0x0000, R2
loop    SUBI R1, 0x0004, R1
        LD   R3, 0x0004(R1)
        LD   R4, 0x0044(R1)
        MUL  R3, R4, R3
        ST   R3, 0x0084(R1)
        ADD  R2, R3, R2
        BRGE R1, loop
```



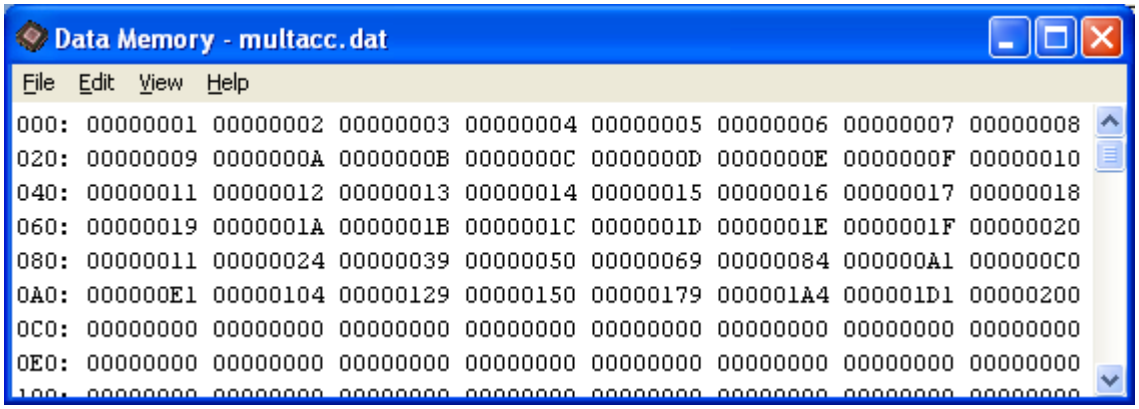
```
int A[16]={0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,\
           0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,0x10};

int B[16]={0x11,0x12,0x13,0x14,0x15,0x16,0x17,0x18,\
           0x19,0x1A,0x1B,0x1C,0x1D,0x1E,0x1F,0x20};
int C[16];

void main(void)
{
    int s=0;
    int i;

    for(i=15; i>=0; i--)
    {
        C[i] = A[i]*B[i];
        s += C[i];
    }
}
```

Optymalizacja kodu – projekt ‘multacc’

Po wykonaniu programu zawartość pamięci wygląda następująco:

- adresy 0x00.. 0x3C tablica A (bez zmian)
- adresy 0x40.. 0x7C tablica B (bez zmian)
- adresy 0x80.. 0xBC tablica C (nowa, obliczona jako $C_i = A_i * B_i$)

a rejestr R2 = 0x00000E58.

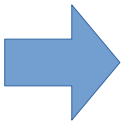


Optymalizacja kodu – projekt ‘multacc’

Zadanie: wykonać optymalizację kodu z projektu ‘multacc’
metodą podwójnego rozwinięcia pętli (two-fold **loop-unrolling**).
Metoda ta polega na redukcji liczby cykli i duplikacji instrukcji wewnątrz pętli.

Zasada działania metody polega na zwiększeniu liczby niezależnych instrukcji (czyli wątków obliczeniowych), które wykorzystuja różne rejestry i nie mają wzajemnych konfliktów oraz odpowiednie ich uszeregowanie (wymieszanie).
W efekcie program wynikowy będzie większy, ale czas jego wykonywania będzie krótszy.

```
void main(void)
{
    int s=0;
    int i;

    for(i=15; i>=0; i--)
    {
        C[i] = A[i]*B[i];
        s += C[i];
    }
}
```



```
void main(void)
{
    int s=0;
    int i;

    for(i=15; i>=0; i-=2)
    {
        C[i]   = A[i]*B[i];
        C[i-1] = A[i-1]*B[i-1];
        s += C[i];
        s += C[i-1];
    }
}
```

Poprawnie zaimplementowana metoda rozwinięcia pętli pozwala na zmniejszenie czasu wykonywania z 244 do **116 cykli** (bez forwarding’u) i oczywiście uzyskanie tych samych wyników, tj. $C_i = A_i * B_i$ oraz R2 = 0x0E58.