Projekt SYKO (Zima 2015) **Ostatnia aktualizacja: 2015.10.12**

Założenia ogólne

1.Punktacja (w całości 50pkt)

20pkt - implementacja
20pkt - testowanie
10pkt - dokumentacja

2.Grupa - trzy osobowa - nie może zmieniać tematu podczas pracy (przypisanie tematu do osób). W przypadkach spornych - decyduje prowadzący zajęcia projektowe.

Założenia szczegółowe

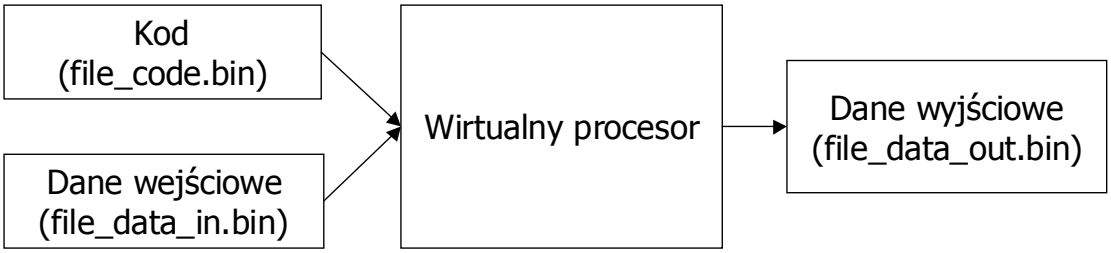
1.Implementacja wirtualnego procesora.

Implementacja wirtualnego procesora powinna być zapisana w języku "C", w jednym z następujących środowisk programistycznych:

GCC -działającym "pod system" Linux (Debian, Ubuntu, Fedora, Cygwin)
MinGW -darmowy kompilator C (na bazie gcc), dla środowiska win32:
http://www.mingw.org,
http://sourceforge.net/projects/mingw/files/
zalecana wersja - nie starsza niż 3.2.3, opis co trzeba pobrać i jak instalować jest na stronie: *http://www.mingw.org/download.shtml*

Implementacja powinna być pozbawiona wszelkich konstrukcji właściwych dla C++ czy C#. Dodatkowo nie wolno używać elementów interfejsu z użytkownikiem (jakakolwiek grafika lub jakiekolwiek „ramki" w trybie tekstowym).

Na poniższym rysunku widać przepływ informacji wejściowych/wyjściowych oraz umiejscowienie wirtualnego procesora:



Jak widać na powyższym rysunku kod emulowanej aplikacji testowej (binarna forma) - wykonywany przez wirtualny procesor - ładowany jest z pliku:

```
file_code.bin
```

a dane wejściowe dla tej aplikacji testowej, wczytywane są z pliku:

```
file_data_in.bin
```

Po zakończeniu wykonywania aplikacji testowej przez procesor wirtualny zawartość pamięci danych powinna być zapisana do pliku:

```
file_data_out.bin
```

Na bazie analizy zawartości pamięci danych możliwe jest ustalenie czy wirtualny procesor wykonał poprawnie aplikację testową.

W dodatku **A** zawarto szkielet interpretera (podstawowa część wirtualnego procesora) na bazie którego można napisać własny wirtualny procesor.

2.Implementacja - postać źródłowa

Kod źródłowy implementacji wirtualnego procesora powinien być zwięzły i efektywny. Rozumie się przez to że ma być on jak najkrótszy (np.: co liczby linii kodu, liczby znaków) a po kompilacji działać poprawnie i jak najszybciej. Dodatkowo negatywnie o jakości

utworzonej implementacji świadczyć będzie rozrzutność w użytych zasobach wymaganych dla pracy wirtualnego procesora.
Zatem kody źródłowe oddawanego projektu powinny (kolejność odzwierciedla wagę):

-działać poprawnie (zgodnie z opisem wirtualnego procesora),
-dawać się testować (zgodnie z przygotowanymi danymi wejściowymi),
-dawać się ponownie kompilować, a wyniki kompilacji powinien dawać się uruchamiać wielokrotnie (dając za każdym razem oczekiwany rezultat),
-być czytelne i poprawnie skomentowane.

Zalecenia odnośnie dobrego kodowania w języku C, można znaleźć na stronie:
http://galaxy.uci.agh.edu.pl/~worek/Praktyki/standard_kodowania_2007-07-17.pdf

3.Testowanie własnego produktu

Jednym z elementów zadania projektowego jest przemyślane przygotowanie plików testowych. Pliki te mają potwierdzić prawidłowe działanie emulowanych instrukcji poprzez tworzony wirtualny procesor.

3.1.Przygotowanie plików testowych

Zakłada się że każdy plik testowy jest mini-aplikacją testującą wirtualny procesor pod danym kątem (np.: instrukcje dodawania, instrukcje warunkowe, instrukcje logiczne, ...). Zaleca się zatem przemyślenie co i w jakiej kolejności będzie testowane.
W wyniku takich przemyśleń powinno powstać wiele (nieraz bardzo specyficznych) plików testowych oraz - co czasami jest nieodzowne - odpowiadających im plików z danymi ładowanymi do pamięci danych przy uruchomieniu testu (wczytywanych jako „file_data_in.bin").
Testy te powinny sprawdzać zarówno typowe zachowania wirtualnego procesora, jak i warunki brzegowe poszczególnych jego instrukcji.
Dla przykładu testując zachowanie dla instrukcji dodawania, np.: **`ADC R1, R2`** można napisać następujący kod (zakładając że poprzednio przetestowano całościowo poprawność działania instrukcji **`MOV`**):

```
MOV  R1, CONST1                  //R1=CONST1
MOV  R2, CONST2                  //R2=CONST2
MOV  FLAGS,CONST3                //FLAGS=0 (w tym C=0)
ADC  R1, R2                      //R1=R1+R2+C
MOV  GDZIES_DATA1, R1            //MEM[GDZIES_DATA1]=R1
MOV  GDZIES_DATA2, FLAGS         //MEM[GDZIES_DATA2]=FLAGS
```

Dla sprawdzenia wszystkich warunków (zwykłych i brzegowych) test ten trzeba przeprowadzić dla wielu różnych wartości stałych: `CONST1`, `CONST2` oraz `CONST3` - a nie tylko dla jednego „jakoś" wybranego zestawu tych wartości. Dla testowania warunków zwykłych będą to wartości, np.:

```
CONST1      CONST2      CONST3 (bit C)
1           2           0
1           2           1
```

Natomiast dla testowania warunków brzegowych hipotetycznego wirtualnego procesora 4 bitowego - czyli takiego w którym ALU operuje naraz na 4 bitach - należałoby wykonać testy także z wartościami:

```
CONST1      CONST2      CONST3 (bit C)
1           14          0
1           14          1
1           15          0
1           15          1
14          1           0
14          1           1
15          1           0
15          1           1
```

Pliki binarne testujące wirtualny procesor można tworzyć "ręcznie", lecz jest to proces bardzo pracochłonny i podatny na błędy. Innym zalecanym podejściem jest wykorzystanie kompilatorów języka asembler dla testowanego procesora. Takie programy są z reguły dostarczane łącznie z programami narzędziowymi kompilatorów języka C:

Procesor	Pakiet kompilatora	Strona z narzędziami (dla Win32)
MSP430	TI-GCC	*http://www.ti.com/tool/msp430-gcc-opensource (wymagana rejestracja, choć narzędzie jest darmowe)*

3.2.Jak dokumentować testy

Przygotowanie plików z testami to etap wstępny, przeprowadzenie testów to etap właściwy procedury testowej a etap finalny to udokumentowanie że dany test został przeprowadzony i wynik są zgodne z oczekiwaniami.
Podstawowe założenie właściwego etapu testowania, to założenie, że wirtualny procesor ma działać zgodnie z jego dokumentacją. Często twórca takich testów zapomina o bardzo istotnym fakcie, że procesor wykonując to co opisuje dokumentacja nie może wykonać nic więcej (np.: zmodyfikować nieodwracalnie jakąś cząstkę pamięci „na chwilę"). Jednym z elementów testowania jest zatem dowód wykazujący, że procesor wykonał zadany test i nie zmienił się jego stan w niezamierzony sposób (dotyczy to nie tylko jego rejestrów ale i reszty logiki w tym i zawartości jego pamięci).
Aby udokumentować w zwięzły sposób poprawność stanu procesora po zakończeniu danej operacji (lub zestawu operacji) można posługując się narzędziami porównywania plików binarnych sprawdzić stan pamięci danych przed i po danym zestawie operacji – i w odpowiedni sposób zacytować co w zawartości pamięci zostało zmienione. Jednym z podstawowych narzędzi jest „CMP" (narzędzie dostępne w większości dystrybucji systemu Linux, czy pod cygwin), dla dwóch plików „`file_data_in.bin`" i „`file_data_out.bin`" wynik porównania mógłby wyglądać następująco:

```
>cmp file_data_in.bin file_data_out.bin -l
 8 125  61
15  31 101
16 111 101
```

Z takiego raportu widać że różnice są na pozycjach: 8, 15 i 16 oraz widać jak zmieniła się treść. Pierwszy plik zawierał na pozycji 8 znak o kodzie 125 a drugi o kodzie 61, itd. Choć podejście te z początku jest mało przyjazne, ułatwia w efekcie dokumentowanie zmian – pokazując tylko to co się zmieniło.

4.Procedura oddawania projektu

Dla uproszczenia procedury oddawania projektu, prace wynikowe powinny być przygotowane dla środowiska komputera klasy PC. Zaleca się jednak wykorzystanie jednego z systemów wirtualizacji: *VirtualBox, Qemu*.
Przygotowanie takiej wirtualizacji zapobiegnie sytuacjom w którym podczas obrony projektu, nie powiedzie się jego uruchomienie, zarówno postaci wynikowej jak i kompilacja postaci źródłowej (np.: na skutek różnic środowisk kompilacji u studenta i u prowadzącego).
Sugerowane jest także dodatkowo przygotowanie wirtualizacji w taki sposób aby:
-zainstalowany w niej był kompilator (typ1) - niezbędny dla kompilacji implementacji wirtualnego procesora (tu: MinGW/GCC),

-zainstalowany w niej był kompilator (typ2) - niezbędny dla kompilacji aplikacji testujących wirtualny procesor (tu: TI-GCC),

oraz dodatkowo:
-ustawienie karty sieciowej w wirtualnym PC było takie aby pobierany był numer IP z serwera DHCP,
-ustawienie i uruchomienie serwera SSH - dla łatwego przenoszenie plików do/z wirtualizowanego środowiska.

Mimo ułatwień ze strony uczelni, ze względu na problemy licencyjne, wirtualizowany PC powinien mieć zainstalowany system operacyjny dla którego nabywanie licencji nie jest wymagane (Ubuntu/Debian/...).

5.Wirtualny procesor MSP430

Informacje o budowie listy rozkazów procesora MSP430 można znaleźć na stronie:

http://focus.ti.com/lit/ug/slau049f/slau049f.pdf

Należy tutaj zaznaczyć, iż dla danej grupy projektowej podane do zaimplementowania instrukcje mogą występować w różnych trybach adresowania (ang. addressing mode) - obowiązuje zatem implementacja wszystkich ich kombinacji co oznacza, że liczba instrukcji do zaimplementowania może być granicznie równa:

{zadana liczba instrukcji} x {wszystkie tryby adresowania dla danej instrukcji}

W podanym wyżej pliku PDF dla procesora MSP430, w rozdziale 3.4 (strony 3-17...3-75) zawarto sposób kodowania instrukcji i ich znaczenie. Szczególnie przydatna jest tabela na stronie 3-74, pokazano w niej w zwarty sposób kodowanie wszystkich instrukcji.
Warto zwrócić uwagę ze niektóre instrukcje mogą nie występować bezpośrednio w procesorze a być emulowane poprzez inne instrukcje, często bardziej zawiłe np.:

```
BR dst      -> MOV PC, dst
INC dst     -> ADD #1, dst
```

5.1.Testowanie własnej implementacji procesora MSP430

Sposób posługiwania się kompilatorem (z pakietu TI-GCC) dla generowania plików testowych jest następującym:

a)Tworzenie z pliku asemblerowego (plik: main.S), wyniku w postaci pliku ELF:

```
>msp430-elf-as -mmcu=msp430x149 main.S -o main.elf
```

b)Przekształcanie postaci ELF w postać z surowymi binariami – czyli wygenerowanie zawartości pliku: file_code.bin:

```
>msp430-elf-objcopy --output-target binary main.elf file_code.bin
```

c) Weryfikacja procesu kompilacji (de-asemblacja wyniku kompilacji) - przydatne dla celów sprawdzenia poprawności sporządzenia pliku: `main.S`:

```
>msp430-elf-objdump -d main.elf > main.lst
```

W pliku: `main.lst`, zawarta jest ostateczną postać wyniku kompilacji – zalecane jest sprawdzenie tej postaci.

W poniższym listingu zawarto przykład pliku: main.S (prosta pętla i parę podstawień):

```
        .text
        .p2align    1,0
        .global     main
        .type       main,@function
main:
        mov         #(__stack), r1
        mov         r1,r4
        mov         #llo(12345), @r4
        mov.b       #llo(65), 4(r4)
loop1:
        add.b       #llo(1), 4(r4)
        jmp         loop1
```

Dodatek A - Szkielet symulatora (pseudokod)

Założenia dla tego przykładu:
-Kod operacji zapisywany na 4 najstarszych bitach (bity 15..12), słowa 16 bitowego pobieranego jako całość, numer rejestru R1, zapisywany na 4 najmłodszych bitach (bity 3..0), a R2 na bitach 7...8.

```
...
#define      MAX_WARTOSC ...                  //maks. wartość dla domyślnego typu
typedef .... TypDanych;
typedef .... TypKodu;
typedef .... TypAdresu;
...
TypKodu      MEMC[MAX_ADDRESS];               //obszar pamięci kodu
TypDanych    MEMD[MAX_ADDRESS];               //obszar pamięci danych
TypDanych    REJ[MAX_REJESTR];                //deklaracja przechowywania rejestrów
TypDanych    T;                               //zmienna globalna pomocnicza
TypAdresu    PC;                              //zmienna globalna – licznik rozkazów
...
void main(void){
      ...
      Laduj(MEMC, "file_code.bin");           //Ładowanie pamięci kodu z pliku
      Laduj(MEMD, "file_data_in.bin");        //Ładowanie pamięci danych z pliku
      PC=...;                                 //Warunki początkowe PC (RESET)
      ...                                     //Inne inicjacje wirtualnego procesora
      for(;;){
            T=MEMC[PC];                       //T=ID operacji i arg. wbudowanych
            PC+;                              //zwiększenie licznika rozkazów
            switch((T & 0xF000)>>12){         //wyłuskanie właściwego kodu operacji
                  case  ID_ADD_R1_R2:         //instrukcja ADD R1,R2
                        printf("ADD_R1_R2\n");
                        F_ADD1();             //właściwe wywołanie kodu
                                              //implementacji tej operacji
                        break;
                  case  ID_ADD_R1_MEM_R2:     //instrukcja ADD R1,[R2]
                        printf("ADD_R1_MEM_R2\n");
                        F_ADD2();             //właściwe wywołanie kodu
                                              //implementacji tej operacji
                        break;
                  ...
                  default:
                        printf("Nieznana instrukcja
                              (PC=%lx, T=%lx)!\r\n", PC, T);
            }
      }
      Zapisz("file_data_out.bin");     //zapisz zawartość pamięci danych do pliku
}
```

Implementacje operacji: `ADD R1,R2` oraz `ADD R1,[R2]`, wyglądałyby następująco:

```
void F_ADD1(void){
  TypDanych R1=T & 0x000F;                   //identyfikacja numeru rejestru arg. 1
  TypDanych R2=(T & 0x00F0)>>4;              //identyfikacja numeru rejestru arg. 2

  if(REJ[R1] + REJ[R2] > MAX_WARTOSC)
     REJ[FLAGS].C=1;                         //uaktualnienie przeniesienia;

  REJ[R1]=(REJ[R1] + REJ[R2]) % MAX_WARTOSC; //właściwe obliczenie
}

void F_ADD2(void){
  TypDanych R1=T & 0x000F;                    //identyfikacja numeru rejestru arg. 1
  TypDanych R2=(T & 0x00F0)>>4;               //identyfikacja numeru rejestru arg. 2

  if(REJ[R1] + MEMD[REJ[R2]] > MAX_WARTOSC)
     REJ[FLAGS].C=1;                          //uaktualnienie przeniesienia

  REJ[R1]=(REJ[R1] + MEMD[REJ[R2]]) % MAX_WARTOSC; //właściwe obliczenie
}
```