Procesory Nios II w układach FPGA (4)

Obsługa przerwań i debuggowanie kodu programu

W wielu aplikacjach z mikrokontrolerem należy obsłużyć jedno lub kilka źródeł przerwań. W artykule przedstawiono sposób obsługi przerwań w procesorach Nios II implementowanych w układach programowalnych firmy Altera. Nowotworzone programy dla mikrokontrolerów nie zawsze działają zgodnie z założeniami. Aby odnaleźć źródło błędnego działania programu należy posłużyć się debuggerem.

Przerwania w procesorach Nios II

Przerwania w procesorach Nios II mogą być obsługiwane dwojako: poprzez wbudowany kontroler przerwań Internal Interrupt Controller (IIC) lub poprzez interfejs służący do dołączania zewnętrznego kontrolera przerwań External Interrupt Controller (EIC). Rodzaj kontrolera przerwań jest wybierany na etapie dodawania procesora do tworzonego systemu cyfrowego w programie SOPC Builder. W dalszej części artykułu omówiony zostanie wyłącznie kontroler IIC, ponieważ kontroler EIC jest dostępny wyłącznie dla procesora Nios II/f.

Wyjątki w kontrolerze IIC są obsługiwane podobnie jak w innych procesorach RISC. Wszystkie wyjątki, w tym przerwania sprzętowe, są obsługiwane przez kod programu umieszczony pod specjalnym adresem w pamięci programu. Obsługą wyjątków zajmuje się biblioteka Altera HAL. Oznacza to, że nie ma tutaj mowy o wektorze przerwań, więc wybór funkcji obsługi przerwania jest dokonywany programowo.

Architektura kontrolera IIC umożliwia obsługę do 32 sprzętowych źródeł przerwań. Rdzeń procesora ma 32 wejścia przerwań oznaczone `irq0...irq31` dla każdego ze źródeł przerwania. Oprogramowanie procesora może indywidualnie włączać lub włączać każde ze źródeł przerwania poprzez modyfikację odpowiadających im bitów w rejestrze kontrolnym `ienable`. Dodatkowo w rejestrze `status` znajduje się bit globalnego zezwolenia na przerwania (PIE).

Do obsługi kontrolera przerwań służą dedykowane funkcje HAL API:

`alt_ic_isr_register()` - służy do definiowania funkcji obsługi danego przerwania,

`alt_ic_irq_enable()` - włącza konkretne przerwanie,

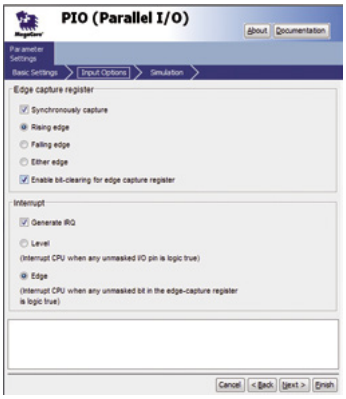
`alt_ic_irq_disable()` - wyłącza konkretne przerwanie,

`alt_ic_irq_enabled()` - służy do sprawdzania, czy przerwanie o podanym numerze jest włączone,

`alt_irq_disable_all()` - służy do wyłączanie wszystkich maskowalnych przerwań,

`alt_irq_enable_all()` - włącza wszystkie przerwania wyłączone przez funkcję `alt_irq_disable_all()`.

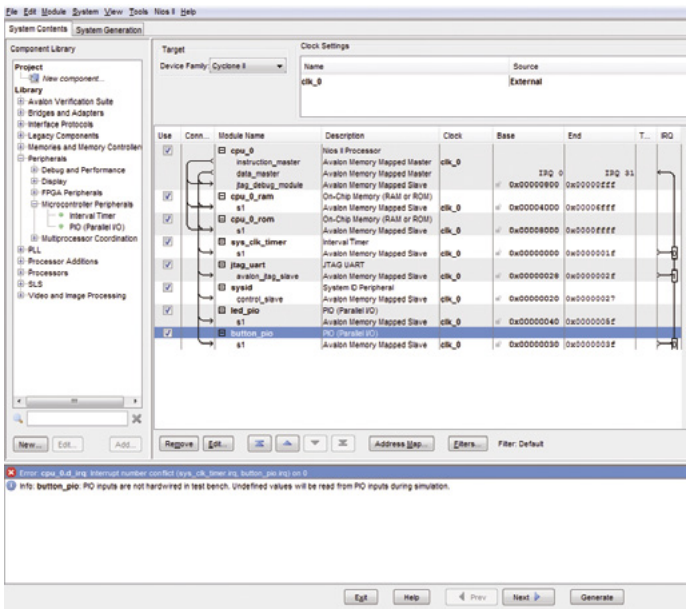
Pierwsze cztery funkcje są obsługiwane przez sterowniki programowe danego kontrolera przerwań. W przypadku kontrolera IIC funkcje te są obsługiwane przez HAL. Pozostałe trzy funkcje są zawsze obsługiwane przez



Rys. 31. Ustawienie modułu button_pio

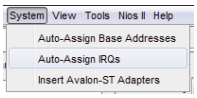
HAL. Funkcje te należą do nowej, ulepszonej biblioteki HAL API (w literaturze *enhanced API*). Obecnie wspierane są funkcje z obydwu wersji biblioteki API, ale firma Altera zaleca stosowanie wyłącznie nowej biblioteki.

Funkcja `alt_ic_isr_register()` przyjmuje następujące parametry:



Rys. 32. Widok połączeń źródeł przerwań z procesorem Nios II

ic_id typu alt_u32, jest numerem ID kontrolera przerwań, który zdefiniowany jest w pliku nagłówkowym system.h. W przypadku braku kontrolera EIC w systemie parametr ten jest ignorowany,



Rys. 33. Automatyczne funkcje programu SOPC Builder

irq typu alt_u32, jest numerem przerwania, przypisanym do danego bloku IP w systemie cyfrowym,

isr jest wskaźnikiem do funkcji obsługi przerwania,

isr_context jest wskaźnikiem do typu void przekazywanym podczas wywołania funkcji przy obsłudze przerwania, który wskazuje do struktury danych zdefiniowanej przez programistę,

ostatnia zmienna flags jest zarezerwowana do przyszłych zastosowań.

Funkcja obsługi przerwań została zadeklarowana jako: typedef void (*alt_isr_func) (void* isr_context);.

Listing 6) Obsługa przerwań procesorów Nios II

```
////////////////// część A ////////////////////////////////////////////////
#include "system.h"
#include "alt_types.h"
#include "altera_avalon_pio_regs.h"
#include <unistd.h>
#include "sys/alt_irq.h"

// Zmienne globalne
volatile alt_u32 edge_capture = 0;
volatile alt_u32 przyciski = 0;

/**
 * Funkcja obsługi przerwania przycisków
 */
#ifdef ALT_ENHANCED_INTERRUPT_API_PRESENT
        static void button_irq(void* context)
#else
        static void button_irq(void* context, alt_u32 id)
#endif
{
        // pobierz kontekst
        volatile alt_u32* edge_capture_ptr = (volatile alt_u32*) context;

        // zapisz wartość
        *edge_capture_ptr = IORD_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE);

        // Wyzerowanie odczytu stanu przycisków
        IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0xFF);

        // ustaw lub wyczyść odpowiedni bit
        if(przyciski & edge_capture){
                // ustaw
                przyciski = przyciski & (~edge_capture);
        } else {
                // wyczyść
                przyciski = przyciski | edge_capture;
        }
}
////////////////// część B
////////////////////////////////////////////////
/**
 * Funkcja inicjalizująca dla modułu przycisków
 */
static void init_buttons()
{
        // wskaźnik do kontekstu musi być typu void*
        void* edge_capture_ptr = (void*) &edge_capture;

        // włączenie przerwań wszystkich 4 przycisków
        IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTON_PIO_BASE, 0xf);

        // wyczyszczenie rejestru przechwytywania zbocza
        IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0x0);

        // zdefiniowanie obsługi przerwania
        #ifdef ALT_ENHANCED_INTERRUPT_API_PRESENT
        alt_ic_isr_register(BUTTON_PIO_IRQ_INTERRUPT_CONTROLLER_ID,
                            BUTTON_PIO_IRQ,
                            button_irq,
                            edge_capture_ptr, 0x0);
        #else
                alt_irq_register( BUTTON_PIO_IRQ,
                            edge_capture_ptr,
                            button_irq);
        #endif
}

////////////////// część C
////////////////////////////////////////////////
// funkcja główna programu
int main()
{
        alt_u32 led = 0;
        alt_u32 licznik = 0;

        init_buttons();

        while(1)
        {
                licznik++;
                led = ((licznik << 4) & 0xF0) | (przyciski & 0x0F);
                IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, ~led);
                usleep(100000);
        }
        return 0;
}
```

Aplikacja z obsługą przerwań

Źródłem przerwań w przykładowym programie będzie blok PIO obsługujący przyciski, który był opisany w poprzedniej części kursu. W projekcie sprzętowym systemu cyfrowego należy zmodyfikować moduł button_pio w programie SOPC Builder, tak aby przerwanie było generowane dla narastającego zbocza sygnału (**rysunek 31**).

Zmodyfikowany system cyfrowy powinien wyglądać jak na **rysunku 32**. W widoku połączeń bloków systemu cyfrowego są zaznaczone również źródła przerwań wraz z przypisanym numerem przerwania. Dla projektowanego systemu cyfrowego wszystkie źródła przerwań prowadzą do procesora cpu_0. Po dodaniu nowego źródła przerwania program ustawia mu domyślny numer 0, dlatego też w oknie informacyjnym pojawi się błąd o konflikcie numeracji przerwań. Należy zmienić numer przerwania przy module button_pio na inny. Można to wykonać ręcznie lub wybrać opcję *Auto-assign IRQ's* z menu *System* w celu automatycznego nadania numerów przerwań (**rysunek 33**). Tak przygotowany projekt należy skompilować oraz utworzyć plik dla programatora układu FPGA w programie Quartus II.

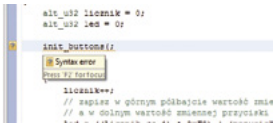
Następnie należy utworzyć nowy projekt aplikacji dla procesora Nios II w programie Nios II EDS. Jako projekt BSP można podać wcześniej utworzony projekt nios_ep_gpio_hal lub utworzyć nowy. Projekt będzie wymagał ponownej kompilacji, gdyż zmieniony został projekt sprzętowy.

Na **listingu 5** przedstawiono sposób obsługi przerwań w procesorze Nios II. Na list. 5a przedstawiono dołączane pliki nagłówkowe do programu, z których ważniejszy to: "sys/alt_irq.h". Zawiera on definicje funkcji obsługi przerwań dostarczanych przez firmę Altera. Zdefiniowane zostały też dwie zmienne globalne:

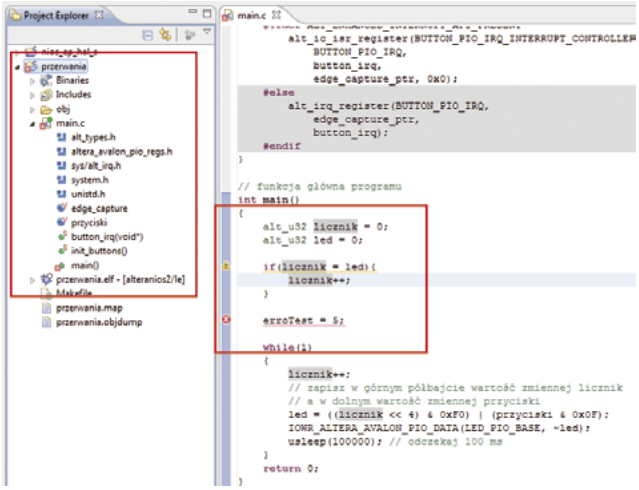
edge_capture – która przechowuje odczytany stan rejestru bloku button_pio, w którym zapisywane są przechwycone zbocza.

przyciski – w tej zmiennej zapisywany jest stan (włączony/wyłączony) danego przycisku. Zmienna ma kwalifikator typu volatile, aby uniknąć ewentualnej optymalizacji kompilatora.

Jak wspomniano wcześniej, funkcje API obsługi przerwań mogą występować w dwóch formach: rozszerzonej oraz przestarzałej (*ang. Deprecated*). Biblioteka rozszerzona jest niedostępna w starszych wersjach



Rys. 34. Zaznaczanie błędów składni w trakcie edycji kodu programu

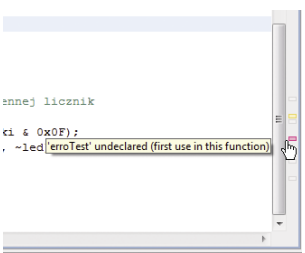


Rys. 35. Zaznaczenie błędów kompilacji w edytorze Nios II EDS

środowiska Nios II EDS, z tego względu na list. 5a definicję funkcji `button_irq` do obsługi umieszczono w deklaracji warunkowej preprocesora. W zależności od wersji środowiska i dostępnej wersji biblioteki obsługi przerwań wybierany jest poprawny sposób deklaracji.

Funkcja obsługi przerwania odbiera jako parametr dowolnie zdefiniowaną przez programistę strukturę danych, tak zwany kontekst (zmienna `context`). W tym przypadku jest to adres zmiennej `edge_capture`. Przy definiowaniu wskaźnika do tej zmiennej należy dokonać rzutowania typu, gdyż zmienna contex będzie zawsze wskaźnika do typu void. Odczytana wartość rejestru EDGE_CAP jest zapisywana do zmiennej edge_capture. Zmienna ta jest następnie używana do zmiany bitów w zmiennej `przyciski`. Bity 3...0 tej zmiennej odpowiadają za włączenie lub wyłączenie odpowiadającej im diody LED. Przy każdym wciśnięciu przycisku odpowiedni bit naprzemiennie ustawiany lub zerowany. Zmienna `edge_capture` została użyta jako maska do ustawiania lub zerowania bitów zmiennej `przyciski`.

Na **listingu 6** w części B przedstawiono funkcję inicjalizującą obsługę przerwania modułu PIO obsługującego przyciski. W funkcji



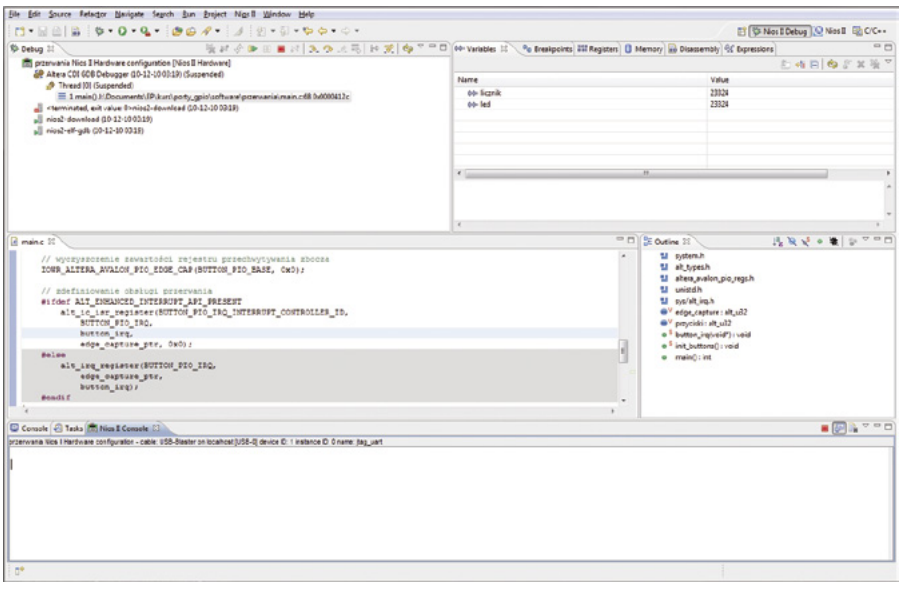
Rys. 36. Nawigacja w edytorze kodu źródłowego

tej ustawiana jest maska przerwań, tak aby wciśnięcie dowolnego przycisku wywoływało przerwanie. Zerowany też jest rejestr `edge_capture`. Na samym końcu rejestrowana jest funkcja obsługi przerwania przycisków. Podobnie jak w przypadku definicji funkcji obsługi przerwania rejestrowanie jest różna w zależności od wersji biblioteki API, z której korzysta program.

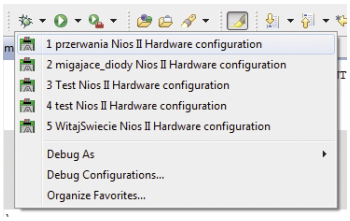
W funkcji głównej (list. 6 część C) wywoływana jest funkcja inicjalizacji przycisków. W głównej pętli programu odczytywany jest stan przycisków ze zmiennej globalnej i jest on używany do włączania odpowiadających im diod LED. Zmienna globalna `przyciski` została zdefiniowana z kwalifikatorem typu volatile, aby jej wartość była odczytywana za każdym razem z przydzielonej komórki pamięci.

Wykrywanie błędów

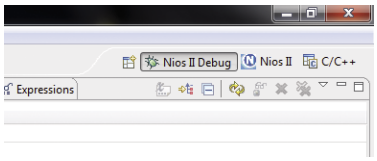
Nieodłącznym procesem przy tworzeniu oprogramowania jest debugowanie kodu programu. Błędy w kodzie programu można podzielić na błędy składniowe, które są wykrywane przez kompilator oraz na na błędy logiczne, które powodują działanie programu niezgodne z założonym. Edytor środowiska Nios EDS w trakcie wpisywania instrukcji zaznacza błędy składni (**rysunek 34**). Po najechaniu kursorem myszy nad podkreśloną przez edytor linijkę kodu pojawia się informacja o rodzaju błędu. Przed skompilowaniem kodu programu, błędy składni są zaznaczone żółtym kolorem podkreślenia, czyli jako ostrzeżeni. Po uruchomieniu kompilacji błędy uniemożliwiające poprawne utworzenie wynikowego programu, w tym błę-



Rys. 39. Widok perspektywy debuggera



Rys. 37. Menu uruchamiania debuggera



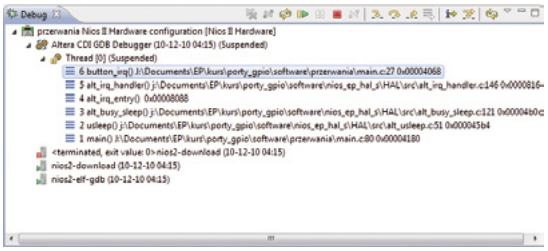
Rys. 38. Przyciski przełączania pespektyw

dy składni, są zaznaczane kolorem czerwonym z ikoną błędu. Kolorem żółtym zaznaczane są możliwe błędy, jak na przykład przypisanie wartości w instrukcji warunkowej if (**rysunek 35**). Dodatkowo, w liście plików w eksploratorze projektów, czerwoną ikoną zaznaczane są pliki, w których wystąpiły błędy.

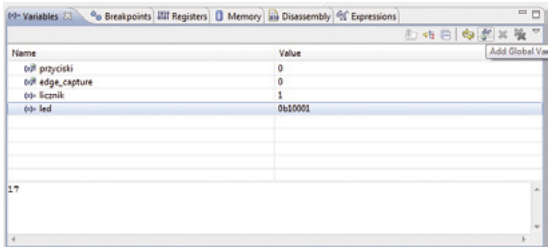
Przydatną funkcjonalnością edytora środowiska Eclipse, na bazie którego jest zbudowany Nios EDS, jest zaznaczanie po prawej stronie okienka edytora miejsca wystąpienia błędów w kodzie programu. Po najechaniu kursorem myszy wyświetlana jest informacja o rodzaju błędu, natomiast po kliknięciu program ustawia kursor edytora w linijce, w której występuje dany błąd (**rysunek 36**).

Do wykrywania błędów logicznych oraz podglądania trybu wykonywania kodu programu służy debugger. W programie Nios II EDS do skompilowania i programu dla debuggera oraz uruchomienia sesji debuggera służy opcja Debug as z menu Run (ikona). Menu Debug As ma analogiczne opcje co menu uruchamiania programu Run As. Jeżeli program był już wcześniej uruchamiany, to na liście dostępnych projektów będzie widniał projekt przerwania (**rysunek 37**).

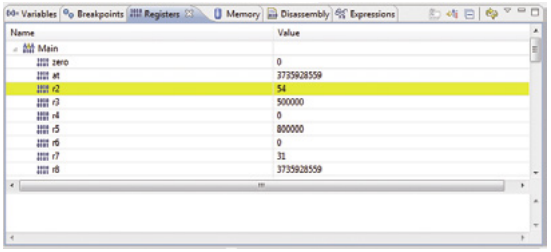
Po uruchomieniu debuggera program wyświetli zapytanie o przełączenie tak zwanej per-



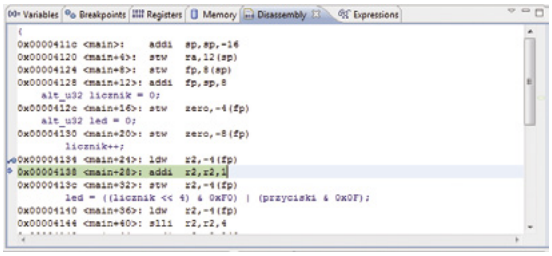
Rys. 40. Główne okno podglądu wykonywanego kodu w debuggerze



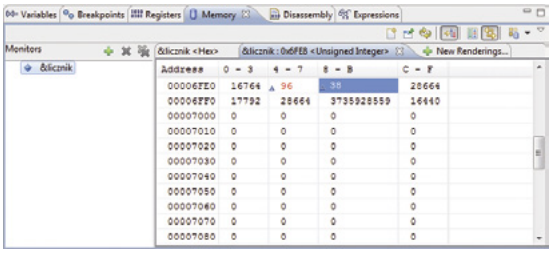
Rys. 41. Okno podglądu zmiennych



Rys. 42. Okno podglądu rejestrów procesora Nios II



Rys. 43. Okno podglądu asemblera procesora Nios II



Rys. 44. Okno podglądu zawartości pamięci

spektywy na Nios II Debug. Należy potwierdzić przełączenie widoku perspektywy oraz ewentualnie zaznaczyć opcję o niepowiadamianiu w przyszłości o jej zmianie. Perspektywy w środowisku Eclipse to nic innego jak zdefiniowany układ okien narzędziowych programu. Jest to przydatne, gdyż niektóre z okien narzędziowych są używane tylko w specyficznych momentach, jak na przykład przy debugowaniu kodu programu. Do przełączania perspektyw służą przyciski zlokalizowane w prawym górnym rogu okna programu (**rysunek 38**).

Widok perspektywy debuggera przedstawiono na **rysunku 39**. Jest on podzielony na trzy główne części: górną, w której zgrupowano okna narzędziowe kontroli wykonywania kodu programu oraz podglądania stanu procesora, środkową z kodem źródłowym oraz dolną w której umieszczono widok konsoli procesora Nios II oraz konsoli komunikatów.

W oknie narzędziowym Debug (**rysunek 40**) umieszczono kontrolki do sterowania procesem wykonywania kodu programu:

- Resume (F8) – wznawianie wykonywania kodu programu
- Suspend – wstrzymywanie,
- Terminate (Ctrl+Z) – zakończenie pracy debuggera,
- Restart – zerowanie procesora do stanu początkowego,
- Step Into (F5) – wykonanie instrukcji znajdującej się pod kursorem z ewentualnym wejściem w ciało funkcji,
- Step Over (F6) – wykonanie instrukcji znajdującej się pod kursorem bez wchodzenia w ciało funkcji,
- Step Return (F7) – wykonanie funkcji do końca oraz wyjście z niej.

W oknie tym wyświetlone są wszystkie uruchomione wątki programu dla wszystkich obserwowanych procesorów, gdyż debugger Nios II umożliwia debugowanie systemów wieloprocesorowych. Ponieważ opisywany program nie jest uruchomiony w systemie operacyjnym µC/OS-II, to w widoku na rys. 40 widoczny jest tylko wątek Thread[0]. Przydatną funkcją jest wyświetlanie poziomu zagnieżdżenia wywoływanych funkcji.

Pozostałymi oknami narzędziowymi są:

Variables – wyświetla zmienne widoczne w danym kontekście (**rysunek 41**),

Breakpoints – wyświetla listę zdefiniowanych pułapek,

Registers – podgląd rejestrów procesora (**rysunek 42**),

Disassembly – widok kodu asemblera skompilowanego programu (**rysunek 43**),

Expressions – widok zdefiniowanych wyrażeń oraz obserwowanych zmiennych,

Memory – podgląd pamięci procesora (**rysunek 44**).

Program umożliwia zamianę wartości zmiennych oraz rejestrów, podgląd zmiennym globalnych (trzeba je ręcznie dodać do okna Variables) a także na edycję zawartości pamięci procesora.

Maciej Gołaszewski
golaszewski.maciej@gmail.com