Inleiding

Om volledig te begrijpen hoe een computerprogramma op computerhardware draait, is niet alleen kennis vereist van de computerarchitectuur, maar ook van de machinetaalinstructies die op deze architectuur uitgevoerd worden.

In het theoriedeel van deze week heb je:

- de ARM Cortex processor familie leren kennen;
- een vereenvoudigde versie van de ARMv7-M architectuur, de LEGv7-M architectuur, leren kennen;
- een vereenvoudigde versie van de Thumb instructieset, de Pinky instructieset, leren kennen;
- assembleertaalprogramma's leren schrijven in Pinky;
- Pinky assembly om leren zetten naar machinecode (encoden);
- Pinky machinecode om leren zetten naar assembly (decoden).

Deze informatie is terug te vinden in De LEGv7 architectuur en de Pinky instructieset[1]. Neem dit document (nogmaals) goed door voordat je aan deze practicumopdrachten gaat beginnen.

Je leert in het praktische deel van deze week:

- Pinky machinecode uit te voeren op het STM32F411E-DISCO ontwikkelbord;
- Pinky assembly code uit te voeren op het STM32F411E-DISCO ontwikkelbord;
- de executietijd van een stukje Pinky assembly code te bepalen en te meten op het ontwikkelbord;
- de leds op het ontwikkelbord aan te sturen via de GPIO peripheral;
- een enkele bit in een I/O-register te lezen of te schrijven met behulp van 'bit-banding';
- Pinky assembly programmadelen aan te roepen vanuit C en vice versa.

[1] https://bitbucket.org/HR_ELEKTRO/rts10/wiki/LEGv7/LEGv7-Pinky_ebook.pdf

Opdrachten week 1 – Inleiding microcontrollerarchitectuur en STM32F411E-DISCO

Blinky in Pinky machinetaal

Een programma om een ledje te laten knipperen is vaak het eerste voorbeeld dat gebruikt wordt om een embedded systeem te leren programmeren, vergelijkbaar met het bekende “Hello World” programma dat bij het leren programmeren op een pc vaak als eerste voorbeeld wordt gebruikt. Zo’n programma dat een ledje laat knipperen wordt vaak “Blinky” genoemd.

1.1 In deze eerste opdracht ga je een programma dat ledjes laat knipperen programmeren in Pinky machinecode. Je maakt daarbij gebruik van het STM32F411E-DISCO ontwikkelbord[2] en STM32CubeIDE[3] maar je maakt *geen* gebruik van een compiler of assembler. In principe willen we de machinecode zelf inladen in het geheugen van de processor. Dit is echter niet mogelijk in STM32CubeIDE omdat we het geheugen pas kunnen aanpassen nadat de debugger is opgestart. Om die reden hebben we wel een klein projectje nodig waarmee we de debugger kunnen starten.

A Download het project opdr_1_1.zip.

B Importeer dit project in STM32CubeIDE met de menu-optie File › Import. . . , General › Projects from Folder or Archive , Next , Archive , selecteer het bestand `opdr_1_1.zip`, Open en Finish .

C Open in de “Project Explorer” het mapje `opdr_1_1.zip_expanded` ▸ `Opdr_1_1` ▸ `Core` ▸ `Src`, zie figuur 1.

```
opdr_1_1.zip_expanded
  opdr_1_1
    Includes
    Core
      Src
        main.s
    STM32F411VETX_RAM.ld
```

Figuur 1: De bestanden in het project `opdr_1_1.zip`.

[2] https://www.st.com/en/evaluation-tools/32f411ediscovery.html

[3] https://www.st.com/en/development-tools/stm32cubeide.html

D Het mapje `Includes` wordt automatisch aangemaakt bij het importeren en kun je negeren. Je ziet dat dit project verder slechts twee bestanden bevat.

Open het bestand `STM32F411VETX_RAM.ld`. Hier vind je een minimaal linkerscript dat er voor zorgt dat de GNU linker (ld)[4] alle code en data in het RAM-geheugen plaatst. Normaal wordt de code in het flashgeheugen van de processor geplaatst, maar omdat we in dit geval de machinecode zelf in willen voeren, is de code in dit project in het RAM geplaatst. Zo'n linkerscript kan behoorlijk ingewikkeld zijn, maar dit eenvoudige script is goed te volgen. Meer informatie over linkerscripts vind je, indien gewenst, in hoofdstuk 3 van de documentatie van de GNU linker (ld).

Open het bestand `main.s`. Hier vind je de Pinky assembly code van een heel eenvoudig programma dat alleen een branch instructie bevat die naar zichzelf springt. Dit programma kan omgezet worden naar een programma dat uitgevoerd kan worden op de microcontroller in het zogenoemde "Executable and Linkable Format" (ELF) formaat[5] m.b.v. de GNU assembler (as)[6]. De woorden die met een punt (`.`) beginnen zijn geen assemblertaalinstructies maar zogenoemde assembler directives. De paars en vet weergegeven directives zijn algemene directives die onafhankelijk zijn van het type microcontroller dat wordt gebruikt. Deze directives worden beschreven in hoofdstuk 7 van de GNU assembler (as) documentatie. Zoek op wat de betekenis is van de directives **`.text`** en **`.global`**. Waarom moet het label `main` globaal gedefinieerd worden?

De andere directives zijn specifiek voor Arm microcontrollers en processoren en worden beschreven in hoofdstuk 9 van de GNU assembler (as) documentatie. Zoek op wat de betekenis is van de directives `.syntax` `.cpu` en `.thumb`. Je wordt wel een paar keer van het kastje naar de muur gestuurd.

E Build nu het project door in de "Project Explorer" met je rechtermuisknop op het project te klikken en de pop-up menu-optie Build Project te kiezen. Je ziet dat er nu een mapje Debug bij is gekomen. Open dit mapje. In dit mapje vind je naast de gegenereerde ELF-file ook de automatisch aangemaakte makefile. STM32CubeIDE maakt onder water gebruik van de GNU make tool[7] om het project te builden.

4 https://sourceware.org/binutils/docs/ld.pdf

5 https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

6 https://sourceware.org/binutils/docs/as.pdf

7 https://www.gnu.org/software/make/manual/make.pdf

Open het bestand `opdr_1_1.list`, dit is een listing van het programma die aangemaakt is door de GNU assembler (as). Je ziet dat het hele programma bestaat uit één machinecode-instructie `e7fe` (een branch instructie die naar zichzelf springt).

Het machinecode programma dat we uit willen voeren bestaat uit 18 instructies gevolgd door een literal section van 16 bytes[8]. De machinecode is gegeven in listing 1. Je gaat deze machinecode nu invoeren in het RAM-geheugen van de microcontroller.

```
2008 4908 6008 2055 0600 4907 6008 2000
4906 6008 230f 031b 4a05 3a01 d1fd 4058
6008 e7f9 3830 4002 0c00 4002 0c14 4002
5854 0014
```

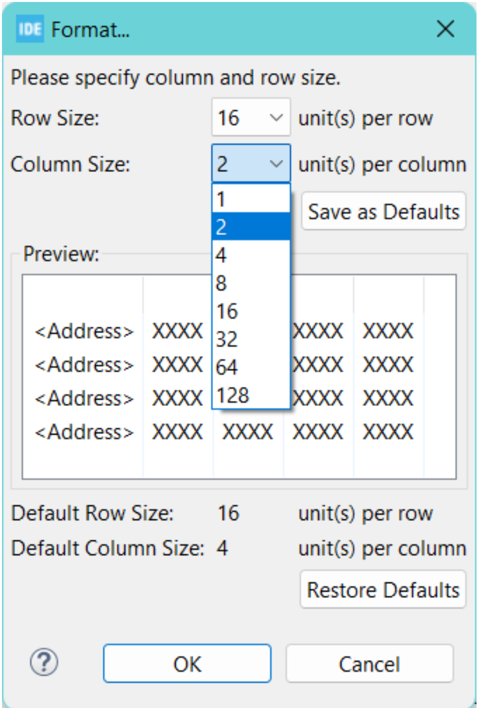
Listing 1: De machinecode van het programma Blinky.

F Start nu de debugger door in de “Project Explorer” met je rechtermuisknop op het project te klikken en de pop-up menu-optie Debug As > 1 STM32 C/C++ Application te kiezen en op OK te klikken. In plaats van dit programma uit te voeren, ga je nu het machinecode programma uit listing 1 invoeren. Selecteer de menu-optie Window > Show View > Memory. Er verschijnt een Memory tabblad in het onderste deelwindow. Klik op het groene plussymbool (Add Memory Monitor), voer het beginadres van het RAM in (`0x20000000`) en klik op OK. Klik nu op het groene plussymbool voor “New Renderings. . . ” en dubbelklik op “Hex Integer”. Klik vervolgens met je rechtermuisknop op het adres `0x20000000` en selecteer de pop-up menu-optie Format. Kies een “Column Size” van 2 en klik op OK, zie figuur 2.

Je ziet dat het adres `20000000` de machinecode E7FE bevat. Dit is de `B.N` instructie uit het programma `main.s`. Voer nu de machinecode uit listing 1 in op adressen `0x20000000` tot en met `0x20000032`. Je kunt alles achter elkaar intypen. Als je geen fouten hebt gemaakt, dan zijn de RAM-adressen `0x20000000` tot en met `0x20000032` nu gevuld met de data die is weergegeven in figuur 3

G Voer dit programma nu uit door op de “Resume” knop of op F8 te klikken. Als het goed is knipperen nu de 4 leds op het STM32F411E-DISCO ontwikkelbord. Stop nu het programma door op de “Terminate” knop of op Ctrl + F2 te klikken.

[8] Als je niet meer weet wat een literal section is, zie dan hier.



Figuur 2: Omdat de Pinky machinecode-instructies 16-bits breed zijn, zetten we de kolomgrootte op 2 bytes.

0x20000000 <Hex> | 0x20000000 : 0x20000000 <Hex Integer> | New Renderings...

Address	0	2	4	6	8	A	C	E
20000000	2008	4908	6008	2055	0600	4907	6008	2000
20000010	4906	6008	230F	031B	4A05	3A01	D1FD	4058
20000020	6008	E7F9	3830	4002	0C00	4002	0C14	4002
20000030	5854	0014	0C14	4002	B0AA	0028	07DA	59F5

Figuur 3: De machinecode is in het RAM van de microcontroller gezet.

1.2 Het eerste deel van het machinecodeprogramma uit listing 1 configureert de GPIO peripheral[9] van de microcontroller. We komen daar later in deze opdracht op terug, zie pagina 9. De GPIO peripheral is zo geconfigureerd dat je de leds kunt beschrijven via bit 12 (groen), bit 13 (oranje), bit 14 (rood) en bit 15 (blauw) van het `GPIOD_ODR` register dat zich op adres `0x40020C14` bevindt. Decodeer nu met behulp van Bijlage C van “De LEGv7 architectuur en de Pinky instructieset” het tweede deel van het machinecode

[9] GPIO = General Purpose Input and Output

programma naar assembly code door de onderstaande tabel 1 verder in te vullen. Het relevante deel van de literal section is in deze tabel ook weergegeven.

Tabel 1: Het tweede deel van het programma Blinky gedecodeerd van machinecode naar assembly code.

adres	hex	binair	assembly code
2000000e	2000	0010 0000 0000 0000	MOVS.N R0, #0
20000010	4906	0100 1001 0000 0110	LDR.N R1, [PC, #24] // [0x2000002c]
20000012	6008	0110 0000 0000 1000	STR.N R0, [R1, #0]
20000014	230f	0010 0011 0000 1111	...
20000016	031b	0000 0011 0001 1011	...
20000018	4a05	0100 1010 0000 0101	...
2000001a	3a01	0011 1010 0000 0001	SUBS.N R2, #1
2000001c	d1fd	1101 0001 1111 1101	BNE.N 0x2000001a
2000001e	4058	0100 0000 0101 1000	EORS.N R0, R0, R3
20000020	6008	0110 0000 0000 1000	STR.N R0, [R1, #0]
20000022	e7f9	1110 0111 1111 1001	...
2000002c	40020c14		.word 0x40020c14
20000030	00145854		.word 0x00145854

1.3 Pas de assembly code in tabel 1 zo aan dat alleen de blauwe led, twee keer zo snel als eerst, knippert, door bestaande instructies en/of data aan te passen[10]. Zet de aangepaste assembly code om naar de bijbehorende machinecode. Test de aanpassingen door ook het RAM-geheugen van de microcontroller aan te passen en het programma uit te voeren. Vergeet ook niet om na het starten van de debugger, de eerste instructie van het programma weer terug te zetten in het RAM-geheugen. Deze instructie wordt natuurlijk bij het laden van het programma overschreven door de **B.N** uit `main.s`. Aanpassingen maken aan een machinecodeprogramma is erg lastig omdat op het moment dat je bijvoorbeeld ergens een instructie toevoegt ook de offsets van veel andere instructies aangepast moeten worden.

[10] Er zijn slechts twee regels in de assembly code die aangepast moeten worden.

1.4 Het zelf decoderen van machinecode-instructies is best lastig. We kunnen dit werk ook overlaten aan een disassembler. In STM32CubeIDE is ook een disassembler ingebouwd. Start de debugger en zorg ervoor dat het gehele machinetaalprogramma weer in het RAM staat. Als je de spanning niet van je bordje hebt gehaald, hoef je alleen de eerste instructie aan te passen. Selecteer nu de menu-optie Window > Show View > Other.... Zoek naar "dis" en open het "Disassembly" window. Vergelijk het gedisassembleerde programma met de assembly code die je in tabel 1 hebt ingevuld. Waarom gaat de disassembler bij het decoderen van de literal section, vanaf adres `0x20000024` de mist in?

Executietijd berekenen en bepalen

In Real-Time systemen is het erg belangrijk om vast te stellen hoeveel tijd het uitvoeren van een bepaald stukje code in beslag neemt. Anders is het niet mogelijk om te voorspellen (of zelfs te bewijzen) dat het real-time systeem al zijn deadlines kan behalen. We komen daar in latere lessen nog uitgebreid op terug. Een voordeel van het gebruik van assembly code is dat we exact kunnen bepalen hoe lang het uitvoeren van een stukje code duurt omdat we exact weten welke instructies uitgevoerd worden. Bij het gebruik van een compiler is de executietijd afhankelijk van de specifieke versie van de compiler en de gekozen optimalisatie-instellingen.

1.5 In deze opdracht ga je met behulp van assembly code een ledje laten knipperen met een frequentie van 2 Hz (0,25 s uit en 0,25 s aan). Hierbij is gegeven dat de klokfrequentie van de microcontroller 16 MHz bedraagt.

A Download het project opdr_1_5.zip.

B Importeer dit project in STM32CubeIDE, build en debug het. Als het goed is, knipperen de leds, alleen nog niet in de juiste frequentie.

C Open het bestand `STM32F411VETX_FLASH.ld`. Zoals je ziet, wordt de code nu in het flashgeheugen geladen en niet in het RAM zoals bij opdracht 1.1.

D Open het bestand `main.s`. Zoals je ziet, is het configureren van de GPIO peripheral 'verborgen' door het in machinecode te programmeren. De loop waarmee de leds afwisselend aan- en uitgeschakeld worden is geschreven in Pinky assembly code en weergegeven in listing 2.
Bepaal nu met behulp van Bijlage A van "De LEGv7 architectuur en de Pinky instructieset" voor elke instructie in de loop hoeveel clock cycles deze in beslag

```
        // loop in Pinky assembler code
loop:   LDR.N   R2, =1234567 // just some random value!
wait:   SUBS.N  R2, #1
        BNE.N   wait
        // toggle leds
        EORS.N  R0, R0, R3
        STR.N   R0, [R1, #0]
        B.N     loop
```

Listing 2: De loop die de leds laat knipperen.

neemt. Let op: De executietijd van de `BNE.N` instructie is 1 clock cycle als er *niet* wordt gesprongen en 1(+P) clock cycles als er *wel* wordt gesprongen. Waarbij P kan variëren tussen 1 en 3.[11]

E Bereken nu de waarde die in `R2` geladen moet worden aan het begin van de loop om de loop exact 4 000 000 clock cycles te laten duren, zodat de leds knipperen met een frequentie van 2 Hz. Voeg eventueel extra `NOP.N` instructies toe, om het exact kloppend te maken.

F Pas het programma aan en test het.

1.6 Om te kunnen bepalen of de leds bij opdracht 1.5 met de juiste frequentie knipperen moet je het aantal clock cycles kunnen meten. De STM32F411VET6 microcontroller heeft in de Cortex-M4 core een zogenoemde Debug Watchpoint and Trace (DWT) unit die dit mogelijk maakt. De documentatie van de microcontroller zelf, die toch maar liefst 844 pagina's beslaat, meldt slechts dat dit mogelijk is, maar vertelt niet hoe dit moet[12].

In de Arm v7-M Architecture Reference Manual[13] is wel te vinden hoe deze DWT unit werkt en hoe die gebruikt kan worden. In paragraaf C1.8.3 wordt uitgelegd dat de teller `CYCCNT` het aantal uitgevoerde clock cycles bijhoudt en niet wordt bijgewerkt als de processor zich in de debug state bevindt (bijvoorbeeld bij een breakpoint). Deze

[11] Het aantal clock cycles dat nodig is om de pipeline opnieuw te vullen, zie pagina 3-4 van de Cortex-M4 Technical Reference Manual. In de praktijk blijkt, bij het uitvoeren van deze instructies op de STM32F411VET6 microcontroller, P altijd 1 te zijn.

[12] https://www.st.com/resource/en/reference_manual/rm0383-stm32f411xce-advanced-armbased-32bit-mcus-stmicroelectronics.pdf#page=823

[13] https://documentation-service.arm.com/static/606dc36485368c4c2b1bf62f#page=719

teller is aanwezig in de STM32F411VET6 microcontroller. De teller kan geactiveerd worden in het DWT_CTRL register, zie paragraaf C1.8.7. Zoek op hoe de CYCCNT teller "enabled" kan worden. De waarde van de teller kan uitgelezen (of aangepast worden) via het DWT_CYCCNT register, zie paragraaf C1.8.8. Zoek de adressen van deze registers op in paragraaf C1.8.6.

A Laad het programma dat je bij opdracht 1.5 hebt geschreven in de debugger.

B Open een memory window op het adres van het DWT_CTRL register, kies voor een "Hex Integer Rendering" en activeer de CYCCNT teller.

C Kies vervolgens voor een "Unsigned Integer Rendering" en reset het DWT_CYCCNT register (je kunt gewoon de waarde nul intikken in de betreffende geheugenlocatie).

D Zet een breakpoint op de eerste instructie van de loop.

E Voer de loop één keer uit en controleer of het DWT_CYCCNT register de waarde 4 000 000 bevat. Pas, indien nodig, de code van opdracht 1.5 aan zodat de loop wel exact 4 000 000 clock cycles duurt.

Het bepalen van de executietijd met behulp van het memory window is een beetje behelpen, maar ik heb geen andere manier kunnen vinden om dit in STM32CubeIDE te doen. Tips zijn welkom!

GPIO peripheral initialiseren

In dit deel van de opdracht ga je de General Purpose Input Output peripheral van de STM32-F411VET6 microcontroller configureren om de vier user leds op het STM32F411E-DISCO ontwikkelbord aan te kunnen sturen. We doen dat in Pinky assembly zodat je exact kunt zien welke instructies nodig zijn om de GPIO pinnen te configureren. Later zul je dit in de hogere programmeertaal C doen of gebruik maken van een speciale configuratietool zodat je helemaal geen code meer hoeft te schrijven.

Om te beginnen moet je opzoeken op welke pinnen van de STM32F411VET6 microcontroller de vier user leds zijn aangesloten. Je vindt dit in UM1842 (Discovery kit with STM32F411VE MCU user manual). Vul tabel 2 verder in.

In RM0383 (STM32F411xC/E advanced Arm-based 32-bit MCUs reference manual) vind je de benodigde informatie over de GPIO peripheral. Alle user leds zijn verbonden met pinnen

Tabel 2: Pinnen waaraan de leds gekoppeld zijn.

User led	Pin
Orange	PD13
Green	...
Red	...
Blue	...

van IO port D, zoals je in tabel 2 kunt zien. Om deze IO port te kunnen gebruiken moet deze voorzien worden van een kloksignaal. In paragraaf 6.3.9 van RM0383 kun je vinden hoe je de IO port D clock kunt enablen. Het `RCC_AHB1ENR` register is onderdeel van de RCC (Reset and Clock Control) module van de STM32F411VET6 microcontroller en heeft een offset van `0x30`. In paragraaf 2.3 van RM0383 kun je het basisadres van deze module vinden. De IO port D clock kan dus enabled worden met de volgende Pinky assembly code:

```
LDR.N    R0, =0x40023800
MOVS.N   R1, #0x08
STR.N    R1, [R0, #0x30]
```

1.7 **A** Download het project opdr_1_7.zip.

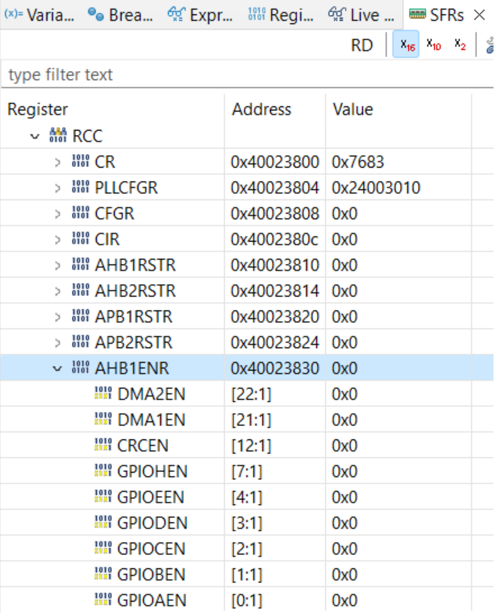
B Voeg onder het label `main:` de bovenstaande assembly code in. Build het project en start de debugger.

C Open nu rechtsboven de tab "SFRs"[14], open de `RCC` module en open daarin het `AHB1ENR` register, zie figuur 4. Zoals je ziet zijn alle kloksignalen van de IO ports disabled na een reset.

D Stap nu door het programma heen en controleer of de bit `GPIODEN` in het `AHB1ENR` register één wordt.

In de bovenstaande assembly code wordt niet alleen de bit `GPIODEN` één gemaakt, maar worden alle andere bits in het `AHB1ENR` register nul gemaakt. Als deze code onderdeel is van een groter programma dat meerdere taken bevat, dan is dit niet gewenst. Misschien is in een ander deel van het programma bijvoorbeeld de IO port A clock al enabled. Om alleen de bit

[14] SFRs staat voor Special Function Registers en in dit tabblad kun je alle IO-registers van de STM32F411VET6 microcontroller bekijken.



Register	Address	Value
RCC		
CR	0x40023800	0x7683
PLLCFGR	0x40023804	0x24003010
CFGR	0x40023808	0x0
CIR	0x4002380c	0x0
AHB1RSTR	0x40023810	0x0
AHB2RSTR	0x40023814	0x0
APB1RSTR	0x40023820	0x0
APB2RSTR	0x40023824	0x0
AHB1ENR	0x40023830	0x0
DMA2EN	[22:1]	0x0
DMA1EN	[21:1]	0x0
CRCEN	[12:1]	0x0
GPIOHEN	[7:1]	0x0
GPIOEEN	[4:1]	0x0
GPIODEN	[3:1]	0x0
GPIOCEN	[2:1]	0x0
GPIOBEN	[1:1]	0x0
GPIOAEN	[0:1]	0x0

Figuur 4: De inhoud van het AHB1ENR register na een reset.

GPIODEN één te maken zonder de overige bits in het AHB1ENR register te beïnvloeden moet gebruik gemaakt worden van een bitwise OR-operatie[15]. De IO port D clock kan dus enabled worden, zonder de overige bits in het AHB1ENR register te veranderen, met de volgende Pinky assembly code:

```
LDR.N    R0, =0x40023800
MOVS.N   R1, #0x08
LDR.N    R2, [R0, #0x30]
ORRS.N   R2, R1
STR.N    R2, [R0, #0x30]
```

1.8 **A** Pas de assembly code in het project opdr_1_7 aan zoals hierboven aangegeven.

B Open nu rechtsboven de tab “SFRs”, open de RCC module en open daarin het AHB1ENR register, zie figuur 4. Enable nu handmatig de IO port A clock door op de waarde van de bit GPIOAEN in het SFRs window te klikken en deze aan te passen.

[15] We gaan ervan uit dat je al weet hoe je met bitwise OR-, AND- en EXOR-bewerkingen individuele bitjes kunt setten, clearen en respectievelijk flippen. Zo niet, dan kun je hier lezen hoe dat werkt.

C Stap nu door het programma heen en controleer of de bit GPIODEN in het AHB1ENR register één wordt en dat de bit GPIOAEN één blijft.

Zoals je ziet kost het één maken van een enkele bit vijf Pinky instructies. Cortex-M4 processoren kunnen optioneel beschikken over een memory map waarin zogenoemde “bit-banding” is opgenomen. Als dit het geval is, dan kunnen bepaalde delen van de memory map ook per bit geadresseerd worden. Elk bit in bijvoorbeeld het AHB1ENR register is dan te lezen of te beschrijven via een ander zogenoemd alias adres. De STM32F411VET6 microcontroller beschikt over “bit-banding”. De benodigde informatie vind je in paragraaf 2.3.3 van RM0383. Iets uitgebreidere informatie vind je in paragraaf 2.2.5 van Cortex-M4 Devices Generic User Guide van Arm[16]. De IO port D clock kan dus enabled worden, zonder de overige bits in het AHB1ENR register te veranderen, met de volgende Pinky assembly code:

```
LDR.N    R0, =0x........ // alias voor de bit GPIODEN in het AHB1ENR register
MOVS.N   R1, #0x01
STR.N    R1, [R0, #0]
```

1.9 **A** Pas de assembly code in het project opdr_1_7 aan zoals hierboven aangegeven. Bereken het benodigde alias adres voor de bit GPIODEN in het AHB1ENR register en vul dit in.

B Open nu rechtsboven de tab “SFRs”, open de RCC module en open daarin het AHB1ENR register, zie figuur 4. Enable nu handmatig de IO port A clock door op de waarde van de bit GPIOAEN in het SFRs window te klikken en deze aan te passen.

C Stap nu door het programma heen en controleer of de bit GPIODEN in het AHB1ENR register één wordt en dat de bit GPIOAEN één blijft.

Nu de IO port D clock enabled is, kun je de IO-pinnen van deze port waarop de user leds zijn aangesloten, zie tabel 2, configureren. Je mag er daarbij vanuit gaan dat de andere pinnen van deze port niet gebruikt worden. Je kunt de benodigde informatie vinden in hoofdstuk 8 van RM0383. In paragraaf 2.3 van RM0383 kun je het basisadres van de GPIOD module

[16] In Table 2-14 “Peripheral memory bit-banding regions” zijn de namen van de “Memory regions” verwisseld. Dus de “address range” 0x40000000 – 0x400FFFFF is de “Peripheral bit-band region” en de “address range” 0x42000000 – 0x43FFFFFF is de “Peripheral bit-band alias”.

vinden. Het programmadeel om de groene en blauwe user leds aan te schakelen kan er bijvoorbeeld als volgt uitzien:

```
        LDR.N   R0, =0x........ // base address for GPIOD
// configure pins for leds in general purpose output mode
        MOVS.N  R1, #0b........
        LSLS.N  R1, R1, #24 // select proper mode
        STR.N   R1, [R0, #0]
// green = on, orange = off, red = off, bleu = on
        MOVS.N  R1, #0b1001
        LSLS.N  R1, R1, #.. // port output data
        STR.N   R1, [R0, #0x14]
```

1.10 **A** Breid de assembly code in het project opdr_1_7 uit met de bovenstaande code. Vul de ontbrekende waarden in.

B Build en debug het project en controleer of alleen de groene en blauwe led aan het einde van het programma aanstaan.

Stel nu dat je later in dit programma de blauwe led uit wilt zetten en de rode led aan wilt zetten, maar dat je niet weet wat de huidige status van de leds is. Je kunt dan niet zomaar een waarde naar het GPIOD_ODR register schrijven omdat je dan ook de overige leds beïnvloedt. Er zijn een aantal verschillende manieren om de blauwe led uit te zetten en de rode led aan te zetten zonder de overige leds te beïnvloeden:

- Via bit-banding de juiste bit in het GPIOD_ODR register te setten en het juiste bit in het GPIOD_ODR register te clearen. Hier zijn zes Pinky instructies voor nodig.
- De waarde van het GPIOD_ODR register inlezen in een register van de processor, een bitwise OR-operatie uitvoeren op dit register om de juiste bit te setten, een bitwise AND-operatie uitvoeren op het register om de juiste bit te clearen en deze waarde weer wegschrijven naar het GPIOD_ODR register. Hier zijn vijf Pinky instructies voor nodig.
- Door gebruik te maken van het “GPIO port bit set/reset register”. Hier zijn slechts drie Pinky instructies voor nodig. De benodigde informatie vind je in paragraaf 8.4.7 van RM0383.

1.11 **A** Breid de assembly code in het project `opdr_1_7` uit met twee assembly instructies[17] waarbij het `GPIOD_BSRR` register gebruikt wordt om de blauwe led uit te zetten en de rode led aan te zetten zonder de overige leds te beïnvloeden.

B Build en debug het project en controleer of alleen de groene en rode led aan het einde van het programma aanstaan.

Grootste Gemene Deler

In de theoriedeel van deze week heb je een functie in assembly code geschreven om de grootste gemene deler (Engels: Greatest Common Divider) te bepalen met behulp van Euclid's algorithm[18]. De C-code is gegeven in listing 3 en de Pinky assembly code is gegeven in listing 4.

```
unsigned int gcd_c(unsigned int a, unsigned int b) {
    while (a != b) {
        if (a > b) {
            a = a - b;
        }
        else {
            b = b - a;
        }
    }
    return a;
}
```

Listing 3: Een C functie om de grootste gemene deler te berekenen.

In listing 5 is een main functie gegeven die geschreven is in Pinky assembly. In dit programma wordt eerst de C functie aangeroepen om de grootste gemene deler van 70 en 130 te berekenen. Vervolgens wordt de assembly functie aangeroepen om hetzelfde te doen. De beide resultaten worden vervolgens met elkaar vergeleken. Als de resultaten ongelijk zijn, dan blijft het programma hangen in de `error:` loop. Als de resultaten wel gelijk zijn, dan blijft het programma hangen in de `ok:` loop.

[17] Je hebt in dit geval maar twee instructies nodig omdat het basisadres van de GPIOD module al in register `R0` is geladen.

[18] https://en.wikipedia.org/wiki/Euclidean_algorithm

```
        .cpu cortex-m4
        .thumb
        .syntax unified
        .text
        .globl gcd
        .thumb_func
gcd:    CMP.N   R0, R1
        BEQ.N   end
        BLS.N   else
        SUBS.N  R0, R0, R1
        B.N     gcd
else:   SUBS.N  R1, R1, R0
        B.N     gcd
end:    BX.N    LR
```

Listing 4: Een Pinky assembly functie om de grootste gemene deler te berekenen.

```
        .cpu cortex-m4
        .thumb
        .syntax unified
        .text
        .globl main
        .thumb_func
main:   MOVS.N  R0, #70
        MOVS.N  R1, #130
        LDR.N   R2, =gcd_c
        BLX.N   R2
        MOVS.N  R4, R0
        MOVS.N  R0, #35
        MOVS.N  R1, #65
        LDR.N   R2, =gcd
        BLX.N   R2
        MOVS.N  R5, R0
        CMP.N   R4, R5
error:
        BNE.N   error
ok:
        B.N     ok
```

Listing 5: Een Pinky main functie die beide gcd functies aanroept en de resultaten vergelijkt.

1.12 In deze opdracht ga je bepalen wat de executietijd is van de C en de Pinky assembly implementatie van de gcd functie. Ook ga je berekenen wat de "Worst Case Execution Time" (WCET) is van de assembly functie.

A Download het project gcd.zip.

B Importeer dit project in STM32CubeIDE, build en debug het. Als het goed is, dan blijft het programma hangen in de `ok:` loop.

C Bepaal de executietijd in clock cycles van de C en de Pinky assembly implementatie van de gcd functie, met de in opdracht 1.6 gebruikte methode.

D Voordat je nu besluit alleen nog maar in assembler te gaan programmeren als de code snel moet zijn, is het verstandig om eerst even te kijken wat de optimizer voor je kan doen. Vul tabel 3 verder in. Het "Optimization level" kun je instellen onder de project "Properties" `C/C++ Buid` ▸ `Settings` ▸ `MCU GCC Compiler` ▸ `Optimization`.

Tabel 3: De executietijd van de verschillende versies van de gcd functie.

taal	optimalisatie	clock cycles
Pinky	–	54
C	`-O0`	151
C	`-O1`	...
C	`-O2`	...
C	`-O3`	...
C	`-Ofast`	...

Wat opvalt is dat `-O3` en `-Ofast` minder snelle code opleveren dan `-O2`. Dat komt wel vaker voor en je moet dus altijd een beetje experimenteren om de snelste code te vinden. Het is zelfs mogelijk om elke individuele optimalisatiemethode aan of uit te zetten, zie paragraaf 3.11 van "Using the GNU Compiler Collection"[19] De compiler kan snellere code produceren dan jij omdat jij alleen een subset van de Thumb instructieset (de Pinky instructieset) gebruikt maar de compiler kan de gehele Thumb instructieset gebruiken.

[19] https://gcc.gnu.org/onlinedocs/gcc-12.1.0/gcc.pdf

E Bereken nu wat de WCET is van de Pinky assembly implementatie van de gcd functie, uitgedrukt in clock cycles. Of anders gezegd: bij welke input duurt deze functie het langst. Je mag er wel vanuit gaan dat beide argumenten groter zijn dan nul want als een van beide nul is loopt de functie vast. Dat geldt trouwens ook voor de C functie. Het zou dus beter zijn om aan het begin van de functie te controleren of één van de parameters nul is en dan de andere parameter als resultaat terug te geven.

F Controleer je berekening door de assembly functie met de worst case argumenten aan te roepen, als je de tijd hebt om daarop te wachten. Bedenk anders een slimme manier om je berekening sneller te verifiëren.

1.13 Bij deze opdracht ga je een snellere versie van gcd implementeren: het "Binary GCD algorithm"[20]. De C code van een recursieve versie is gegeven in listing 6.

A Implementeer de functie `gcd_binary_recursive` in Pinky assembly. De functies `abs_diff` en `min` kun je in C laten staan en vanuit de assembly code aannroepen. Bedenk dat je de operatie `%2` kunt uitvoeren in assembly door alle bits behalve bit 0 nul te maken met behulp van een **`ANDS.N`** instructie. Bedenk dat je de operatie `/2` kan uitvoeren in assembly met behulp van een **`LSRS.N`** instructie. Bedenk dat je de operatie `*2` kan uitvoeren in assembly met behulp van een **`LSLS.N`** instructie.

B Bepaal de WCET van de assembly code van de functie `gcd_binary_recursive`.[21]

[20] https://en.wikipedia.org/wiki/Binary_GCD_algorithm

[21] Mijn implementatie duurt iets minder dan 4000 clock cycles. Dat komt bij een klokfrequentie van 16 MHz overeen met 250 μs.

```c
unsigned int abs_diff(unsigned int a, unsigned int b) {
    if (a > b) {
        return a - b;
    }
    return b - a;
}

unsigned int min(unsigned int a, unsigned int b) {
    if (a < b) {
        return a;
    }
    return b;
}

unsigned int gcd_binary_recursive(unsigned int a, unsigned int b) {
    if (a == 0) {
        return b;
    }
    if (b == 0) {
        return a;
    }
    if (a % 2 == 0) { // a is even
        if (b % 2 == 0) {
            return 2 * gcd_binary_recursive(a/2, b/2); // b is even
        }
        return gcd_binary_recursive(a/2, b); // b is odd
    }
    // a is odd
    if (b % 2 == 0) {
        return gcd_binary_recursive(a, b/2); // b is even
    }
    return gcd_binary_recursive(abs_diff(a, b), min(a, b)); // b ↩
  ↪ is odd
}
```

Listing 6: Een recursieve implementatie gcd_binary_recursive.c.

Antwoorden

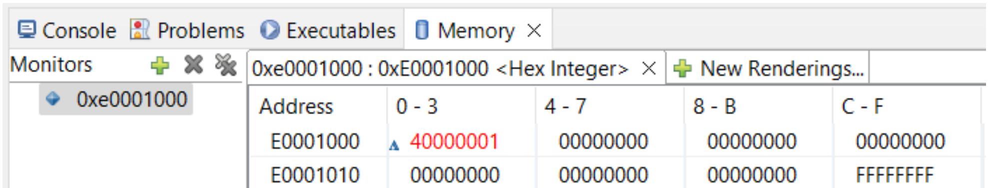
De juiste aanpassingen in het RAM-geheugen voor opdracht 1.3 zijn gegeven in figuur 5.

0x20000000 : 0x20000000 <Hex Integer> | New Renderings...

Address	0	2	4	6	8	A	C	E
20000000	2008	4908	6008	2055	0600	4907	6008	2000
20000010	4906	6008	2308	031B	4A05	3A01	D1FD	4058
20000020	6008	E7F9	3830	4002	0C00	4002	0C14	4002
20000030	5854	000A	2C2A	4002	B0AA	0028	07DA	59F5

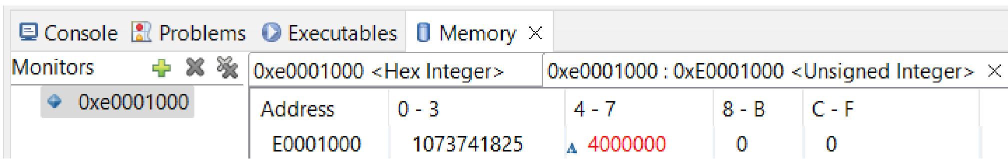
Figuur 5: De machinecode waarbij alleen de blauwe led, twee keer zo snel als eerst, knippert.

De juiste instelling om de `CYCCNT` teller te activeren in het `DWT_CTRL` register bij opdracht 1.6 deelopdracht B is gegeven in figuur 6.



Figuur 6: De `CYCCNT` teller is geactiveerd.

De gewenste waarde in het `DWT_CYCCNT` register, na het eenmalig uitvoeren van de loop bij opdracht 1.6 deelopdracht C, is gegeven in figuur 7.



Figuur 7: Het `DWT_CYCCNT` register bevat de waarde 4 000 000 na het eenmalig uitvoeren van de loop.

De WCET voor de Pinky assembly implementatie bij opdracht 1.12 deelopdracht E kan als volgt berekend worden. In listing 7 is, met behulp van Bijlage A van "De LEGv7 architectuur en de Pinky instructieset", de executietijd bepaald voor de relevante instructies in de functie gcd. De WCET treedt op als de waarde van de eerste parameter in `X0` één is en de waarde van

de tweede parameter in X1 maximaal is. De maximale waarde voor een 32-bits **unsigned int** is $2^{32} - 1 = 4\,294\,967\,295$. De **BLS.N** zal dan steeds springen en de waarde van X1 zal steeds met één verlaagd worden totdat deze waarde gelijk geworden is aan één. De loop in de functie wordt dan het maximale aantal keer uitgevoerd. Deze loop bestaat uit de volgende instructies: **CMP.N**, **BEQ.N** waarbij niet wordt gesprongen, **BLS.N** waarbij wel wordt gesprongen, **SUBS.N** en **B.N** waarbij wel wordt gesprongen. Deze loopt duurt dus $1+1+2+1+2 = 7$ clock cycles. De loop wordt $4\,294\,967\,295 - 1$ keer uitgevoerd. Dit duurt dus $4\,294\,967\,294 \times 7 = 30\,064\,771\,058$ clock cycles.

```
gcd:   CMP.N  R0, R1      // 1
       BEQ.N  end         // 1 (laatste keer +1)
       BLS.N  else        // 2 (worst case als wordt gesprongen)
       SUBS.N R0, R0, R1
       B.N    gcd
else:  SUBS.N R1, R1, R0  // 1
       B.N    gcd         // 2
end:   BX.N   LR          // 2
```

Listing 7: De executietijd van de relevante instructies in de gcd functie.

Als de waarde van X1 één geworden is, worden de instructies **CMP.N**, **BEQ.N** waarbij wel gesprongen wordt en **BX.N** uitgevoerd. Dit duurt dus nog $1+2+2 = 5$ clock cycles. In totaal duurt de functie dus $30\,064\,771\,058 + 5 = 30\,064\,771\,063$ clock cycles. Bij een klokfrequentie van 16 MHz duurt dit 1879,048 s wat overeenkomt met 31,32 min, dus ruim een half uur.

Je kunt deze berekening ook valideren door de waarde in X1 gelijk te stellen aan x. De executietijd t wordt dan:

$$t = (x-1) \times 7 + 5 = 7x - 2 \text{ clock cycles}$$

Als we voor x 100 invullen verwachten we dus een executietijd van 698 clock cycles. Door een breakpoint te zetten op de eerste en de laatste instructie van de assembly functie gcd kun je verifiëren dat elke loop 7 clock cycles kost. Als je vervolgens het eerste breakpoint verwijdert, dan zul je 696 clock cycles na het begin van de functie bij het laatste breakpoint aankomen. Dit is correct omdat de **BX.N** instructie nog uitgevoerd moet worden.

Het juiste bit-banding alias adres bij opdracht 1.9 kun je ook met behulp van de in de documentatie gegeven formule door de assembler zelf uit laten rekenen:

```
LDR.N   R0, =0x42000000 + 0x00023830 * 32 + 3 * 4
MOVS.N  R1, #0x01
```

```
STR.N    R1, [R0, #0]
```