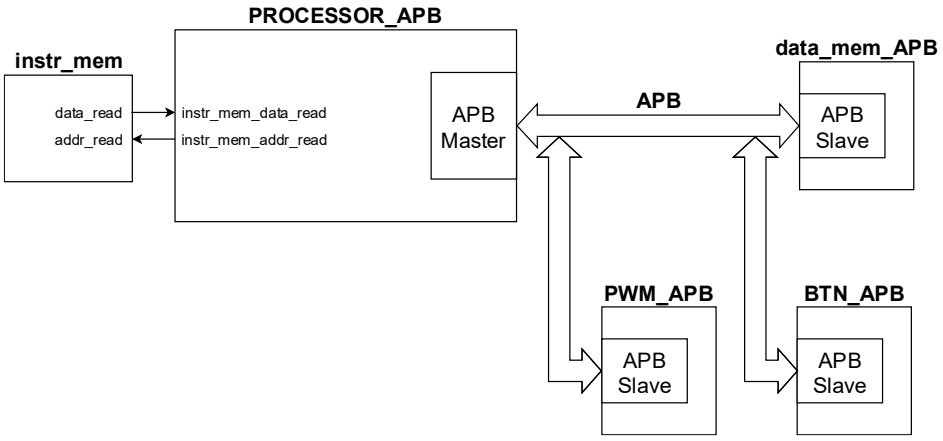
Platformă Sistem APB II

1. Introducere

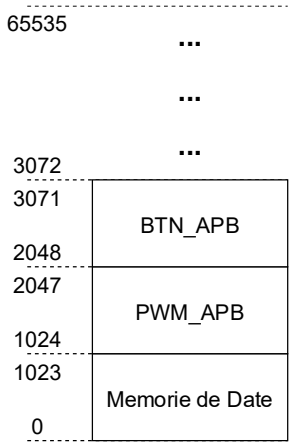
Plecând de la sistemul cu un singur slave (memoria de date) implementat anterior, vom dezvolta un sistem care, pe lângă memoria de date, va avea alte două periferice capabile să comunice cu masterul (PROCESSOR_APB) prin aceeași interfață APB. Bineînțeles, deoarece magistrala APB este comună, nu pot fi accesate mai multe periferice în același timp.

Sistemul dezvoltat anterior se va modifica astfel:



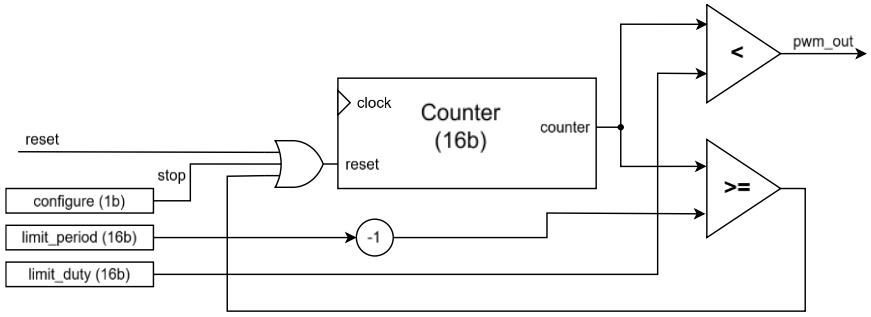
2. Definirea spațiului de adrese

Deoarece toate perifericele sistemului vor fi accesate prin intermediul unor scrieri și citiri într-un spațiu de adrese, trebuie să alocăm fiecăruia un anumit interval. Un astfel de exemplu este prezentat mai jos, unde alocăm fiecărui periferic un spațiu de 1024 adrese. Dacă analizăm valorile capetelor intervalelor de adrese, vom constata că, în cazul sistemului cu 3 periferice, selecția perifericului se poate face cu ajutorul biților [11:10]. Cei mai puțin semnificativi 10 biți ai adresei vor fi folosiți pentru accesarea resurselor din interiorul fiecărui periferic de tip APB.



3. Perifericul PWM_APB

Modulul PWM_APB este un periferic capabil să genereze un semnal PWM cu perioadă și factor de umplere variabile. Configurarea acestuia se va face de către procesor, prin intermediul APB. Schema generală a unui astfel de modul este prezentată mai jos.



Cele 3 registre, *configure*, *limit_period* și *limit_duty* pot fi scrise și citite prin intermediul magistralei APB.

Interfața modulului este prezentată în tabelul de mai jos:

Semnal	Dimensiune	Tip	Descriere
pclk	1	input	Semnalul de ceas
preset	1	input	Semnalul de reset
paddr	10	input	Magistrală de adrese folosită pentru scriere și citire.
psel	1	input	Semnal de selecție a perifericului.
penable	1	input	Semnal de enable. Este 0 în primul ciclu de ceas al transferului și 1 în rest.
pwrite	1	input	Semnal ce arată tipul transferului: 1 pentru scriere și 0 pentru citire.
pwdata	16	input	Magistrală de date de scriere.
prdata	16	output	Magistrală de date de citire. Aceasta va returna conținutul registrului selectat prin adresă atunci când avem un acces de citire valid și 0 în rest.
pready	1	output	Semnal prin care perifericul semnalizează dacă este pregătit să dea curs accesului solicitat de APB Master. În cazul de față, acesta va fi 1 mereu.
pwm_out	1	output	Ieșirea *pwm* a modulului. Aceasta poate fi conectată mai departe la un LED.

Setul de registre:

Semnal	Dimensiune	Tip	Adresă	Descriere
configure	1	W/R	0	Bit[0] – controlează oprirea numărătorului principal al modulului PWM.
limit_period	16	W/R	1	Controlează perioadă semnalului pwm.
limit_duty	16	W/R	2	Controlează factorul de umplere al semnalului PWM.

Observații:

1) Semnalul *prdata* va returna o valoare cu semnificație doar în ultimul ciclu de ceas al unui acces de citire valid (*psel* == 1, *penable* == 1, *pwrite* == 0, *pready* == 1). Atunci când este identificat un acces valid, selecția valorii returnate se va face cu ajutorul adresei *paddr*.

2) În cazul citirii registrului de 1 bit, *prdata* va fi completat cu 0 pentru a extinde valoarea la cei 16 biți ai magistralei de date de citire.

3) Registrele vor fi scrise doar atunci când este identificat un acces valid de scriere (*psel* == 1, *penable* == 1, *pwrite* == 1, *pready* == 1). Selecția registrului ce va scris se face cu ajutorul adresei *paddr*.

4. Perifericul BTN_APB

Perifericul BTN_APB este folosit pentru citirea stării a 4 butoane. Așadar, din punct de vedere al procesorului, resursele acestui periferic se limitează la un registru ce poate fi doar citit. Acest registru va reține la fiecare front crescător al ceasului valoarea butoanelor. Intefața este compusă din urmatoarele semnale:

Semnal	Dimensiune	Tip	Descriere
pclk	1	input	Semnalul de ceas
preset	1	input	Semnalul de reset
paddr	10	input	Magistrală de adrese folosită pentru citire. În cazul acestui periferic, adresa va fi ignorată, deoarece există un singur registru ce poate fi accesat.
psel	1	input	Semnal de selecție a perifericului.
penable	1	input	Semnal de enable. Este 0 în primul ciclu de ceas al transferului și 1 în rest.
pwrite	1	input	Semnal ce arată tipul transferului: 1 pentru scriere și 0 pentru citire.
pwdata	16	input	Magistrală de date de scriere. În cazul acestui periferic, datele de pe aceasta vor fi ignorate.
prdata	16	output	Magistrală de date de citire. Aceasta va returna conținutul registrului de 4 biți extins prin 0 la 16 biți atunci când avem o citire validă și 0 în rest.
pready	1	output	Semnal prin care perifericul semnalizează dacă este pregătit să dea curs accesului solicitat de APB Master. În cazul de față, acesta va fi 1 mereu.
btn_in	4	input	Intrarea de la cele 4 butoane.

Setul de registre:

Semnal	Dimensiune	Tip	Adresă	Descriere
btn_reg	4	R	X	Bit[0] – starea butonului 0 Bit[1] – starea butonului 1 Bit[2] – starea butonului 2 Bit[3] – starea butonului 3

Observație:

Semnalul *prdata* va returna valoare cu semnificație doar în ultimul ciclu de ceas al unui acces de citire valid (*psel* == 1, *penable* == 1, *pwrite* == 0, *pready* == 1).

5. Sistemul APB cu mai multe periferice

Cele trei periferice (Memoria de date, PWM_APB și BTN_APB) vor fi conectate la aceeași magistrală APB.

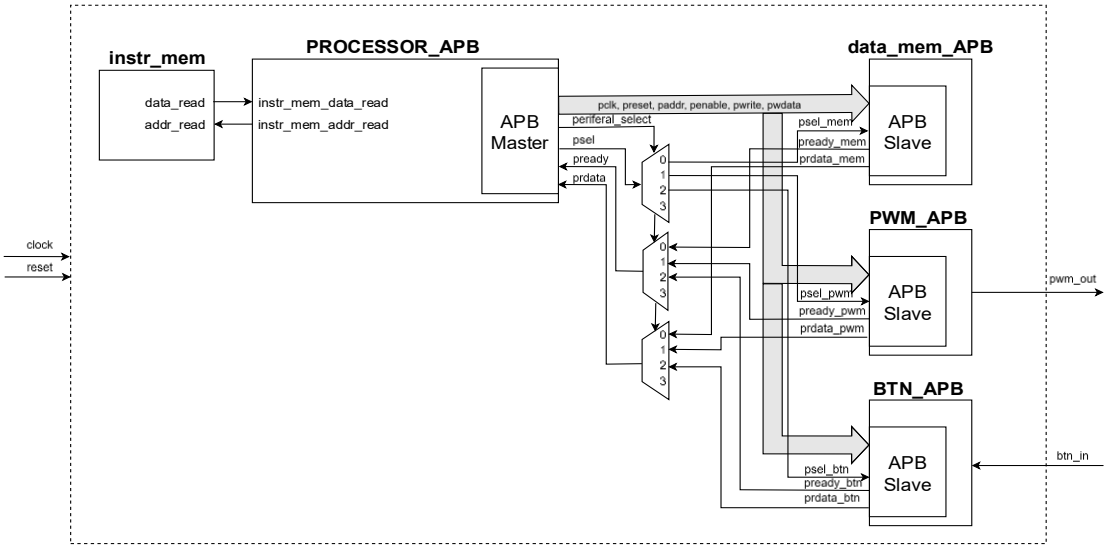
Problema care apare este selecția corectă a perifericului în timpul acceselor de scriere și citire. Pentru a realiza acest lucru, trebuie avute în vedere următoarele:

1) Generarea semnalului *psel* astfel încât un singur periferic să primească *psel* egal cu 1 în timpul accesului de scriere sau citire.
2) Transmiterea către Master (procesor) a semnalului *pready* corespunzător perifericului ce trebuie accesat.
3) Transmiterea către Master (procesor) a datelor *prdata* corespunzătoare perifericului ce trebuie accesat în timpul unui access de citire.

Conform celor descrise în Secțiunea 2, selecția perifericului se poate face cu ajutorul biților [11:10] ai adresei complete. Așadar, pentru a rezolva cele 3 probleme de mai sus trebuie să:

1) Adăugăm o nouă ieșire din Master prin care să transmitem către exteriorul său biții [11:10] ai adresei complete. Ținând cont că pot fi atât accese de scriere, cât și de citire, cei doi biți trebuie să preia biții adresei de scriere, respectiv de citire, în funcție de tipul accesului.
2) Transmitem in paralel către toate perifericele semnalele *pclk*, *preset*, *paddr*, *penable*, *pwrite* și *pwdata*.
3) Adăugăm un demultiplexor care, pe baza celor doi biți, transmite *psel* către un singur periferic.
4) Adăugăm două multiplexoare care, pe baza celor doi biți, transmite către Master (procesor) semnalele *pready* și *prdata* corespunzătoare perifericului ce este accesat.

Schema TOP a circutului devine:



6. Testarea sistemului

Pentru a testa sistemul, pot fi dezvoltate teste care să verifice fiecare periferic în parte, apoi sistemul în ansamblu. Pentru aceasta, vom folosi succesiuni de scrieri și citiri prin APB.

Pentru a realiza testarea, în modulul de test vom genera semnalele *clock* și *reset* și vom inițializa *btn_in* cu o anumită valoare. Ceasul va funcționa pe tot parcursul simulării, în timp ce resetul va fi activ doar la începutul acesteia. Fiind reset sincron, acesta va trebui să fie activ minim o perioadă de ceas.

Dar generarea acestor semnale nu este de ajuns pentru a testa calculatorul elementar implementat. Acesta va trebui să execute un program, o anumită secvență de instrucțiuni. Așadar, memoria de program va trebui inițializată cu șirurile binare ce reprezintă instrucțiunile necesare.

a) Testarea memoriei

Pentru a testa memoria, putem realiza scrieri și citiri la diverse locații de memorie. Pentru aceasta, vom inițializa memoria de program cu următorul program:

```
VALUE_LOAD R0, #14  //R0 = 14
VALUE_LOAD R1, #17  //R1 = 17
VALUE_LOAD R2, #22  //R2 = 22
VALUE_LOAD R3, #92  //R3 = 92
STORE R0, R1        //MEM[R1] = R0
STORE R2, R3        //MEM[R3] = R2
LOAD R4, R1         //R4 = MEM[R1]
LOAD R5, R3         //R5 = MEM[R3]
```

Cele 8 instrucțiuni vor fi stocate în primele 8 locații ale memoriei de program. Extragerea șirurilor binare se face cu ajutorul structurii instrucțiunii prezentate în primul laborator.

De exemplu, pentru următoarele instrucțiuni, vom avea:

VALUE_LOAD R0, #14 //R0 = 14

Opcode (4b)	**Destination (4b)**	**Operand0 (4b)**	**Operand1 (4b)**	**Value (16b)**
1010	**0000**	**0000**	**0000**	**0000_0000_0000_1110**
Opcode-ul instrucțiunii VALUE_LOAD este 10	Destinația este R0, care este codat cu valoarea 0.	Pentru această instrucțiune, acest câmp nu are semnificație și îl putem completa cu 0	Pentru această instrucțiune, acest câmp nu are semnificație și îl putem completa cu 0	Constanta 14, scrisă pe 16 biți

STORE R0, R1 //MEM[R1] = R0

Opcode (4b)	**Destination (4b)**	**Operand0 (4b)**	**Operand1 (4b)**	**Value (16b)**
1101	**0000**	**0000**	**0001**	**0000_0000_0000_0000**
Opcode-ul instrucțiunii STORE este 13	Pentru această instrucțiune, acest câmp nu are semnificație și îl putem completa cu 0	Operand0 este R0, care este codat cu valoarea 0.	Operand1 este R1, care este codat cu valoarea 1.	Pentru această instrucțiune, acest câmp nu are semnificație și îl putem completa cu 0

LOAD R4, R1 //R4 = MEM[R1]

Opcode (4b)	**Destination (4b)**	**Operand0 (4b)**	**Operand1 (4b)**	**Value (16b)**
1110	**0100**	**0001**	**0000**	**0000_0000_0000_0000**
Opcode-ul instrucțiunii LOAD este 14	Destinația este R4, care este codat cu valoarea 4.	Operand0 este R1, care este codat cu valoarea 1.	Pentru această instrucțiune, acest câmp nu are semnificație și îl putem completa cu 0	Pentru această instrucțiune, acest câmp nu are semnificație și îl putem completa cu 0

Așadar, memoria de program trebuie inițializată astfel:

```
initial begin
	mem_instr[0] = 32'b1010_0000_0000_0000_0000_0000_0000_1110; //R0 = 14
	mem_instr[1] = 32'b1010_0001_0000_0000_0000_0000_0001_0001; //R1 = 17
	mem_instr[2] = 32'b1010_0010_0000_0000_0000_0000_0001_0110; //R2 = 22
	mem_instr[3] = 32'b1010_0011_0000_0000_0000_0000_0101_1100; //R3 = 92

	mem_instr[4] = 32'b1101_0000_0000_0001_0000_0000_0000_0000; //MEM[R1] = R0
	mem_instr[5] = 32'b1101_0000_0010_0011_0000_0000_0000_0000; //MEM[R3] = R2

	mem_instr[6] = 32'b1110_0100_0001_0000_0000_0000_0000_0000; //R4 = MEM[R1]
	mem_instr[7] = 32'b1110_0101_0011_0000_0000_0000_0000_0000; //R5 = MEM[R3]
end
```

Observație: Adresele folosite de instrucțiunile LOAD și STORE au fost alese astfel încât să se incadreze în intervalul 0 … 1023, care este alocat memoriei de date, conform Secțiunii 2.

b) Testarea modulului PWM_APB

Pentru a testa modulul PWM_APB, va trebui sa verificăm generarea unui semnal PWM cu un anumit factor de umplere (de exemplu, 50%) și o anumită perioadă. Pentru aceasta, putem folosi următoarea secvență de program:

```
VALUE_LOAD R0, #1024      //R0 = 1024 --> adresa 0 din modulul PWM_APB
VALUE_LOAD R1, #1025      //R1 = 1025 --> adresa 1 din modulul PWM_APB
VALUE_LOAD R2, #25000     //R2 = 25000 --> o valoare aleatorie pentru stabilirea
                          perioadei
VALUE_LOAD R3, #65536     //R3 = 65536 --> 1/2 din intervalul de numărare pentru 50%
STORE R2, R0              //MEM[R0] = R2
STORE R3, R1              //MEM[R1] = R3
```

Observație: În cazul acestui modul, MEM[…] nu reprezintă o locație din memoria de date, ci o locație din spațiul de adrese (aici, corespunde unui registru din PWM_APB).

c) Testarea modulului BTN_APB

Pentru a testa modulul BTN_APB, va trebui sa citim de la orice adresă din intervalul alocat acestui periferic (intervalul 2048 ... 3071). Între timp, din modulul de test, se poate varia valoarea lui *btn_in*. Pentru acest test, putem folosi următoarea secvență de program:

```
VALUE_LOAD R0, #2048      //R0 = 2048 --> adresa 0 din modulul BTN_APB
LOAD R1, R0               //R1 = MEM[R0]
LOAD R1, R0               //R1 = MEM[R0]
LOAD R1, R0               //R1 = MEM[R0]
LOAD R1, R0               //R1 = MEM[R0]
LOAD R1, R0               //R1 = MEM[R0]
LOAD R1, R0               //R1 = MEM[R0]
```

Observație: În cazul acestui modul, MEM[…] nu reprezintă o locație din memoria de date, ci o locație din spațiul de adrese (aici, corespunde registrului din BTN_APB).

d) Testarea sistemului

Propuneți și testați un program care să corespundă următoarei secvențe de pseudo-cod:

```
if(btn_in = 4'b0001)
      pwm_out = 10% duty
else if(btn_in = 4'b0010)
      pwm_out = 25% duty
else if(btn_in = 4'b0100)
      pwm_out = 50% duty
else if(btn_in = 4'b1000)
      pwm_out = 100% duty
else
      pwm_out = 0% duty
```

Observație: Perioada semnalului *pwm_out* poate rămâne constantă pe toată durata simularii.