5TC option BED

Programmation de micro-contrôleurs et périphériques radio

Tanguy Risset, Sebastien Peychet

prénom.nom@insa-lyon.fr

(Précédents contributeurs : Antoine Fraboulet, Antoine Scherrer, Sylvain Geneves, Kevin Marquet, Guillaume Salagnac) Labo CITI, INSA de Lyon, Dpt Télécom

26 septembre 2019

Systèmes enfouis et réseaux de capteurs

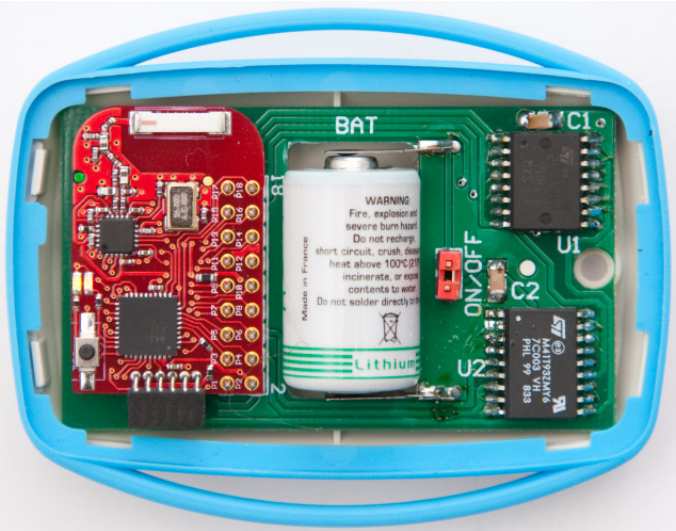
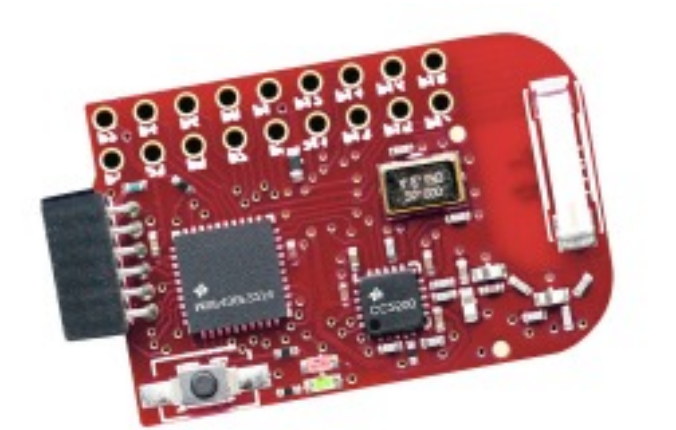


image S.Dalu



- Capteurs de petite taille
- Communication radio
- Mesures et remontées d'alarmes

- Outils et méthodes de développements
- Briques logicielles
  - Modèles de programmation
  - Systèmes d'exploitation
  - Communications radio

Passerelles, interfaces utilisateurs et smartphones

- Le traitement, la consolidation et la mise à disposition des données sont souvent réalisés sur des systèmes plus gros.

- On arrive ici dans le monde des

  «*embedded computing devices*»

Passerelles, interfaces utilisateurs et smartphones

- Systèmes de grande capacité
- Stockage de données et capacités de calculs
- Taille des batteries plus importante

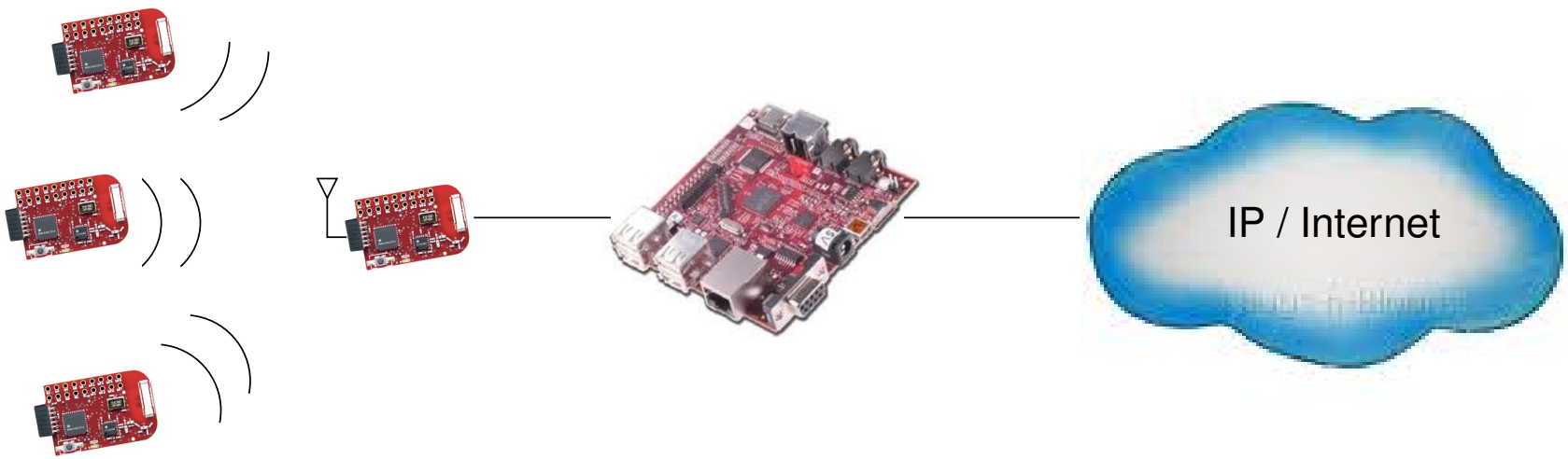
- Outils et méthodes de développements différents

plateformes utilisées dans BED

- **Beagleboard** ($\simeq$ 150€)
- Arm Cortex A8 1 GHz
- DaVinci SoC ARM+DSP
- Puce graphique 3D
- 512 MB of DDR SDRAM
- 4GB SD-Card
- DVI-D, S-Video, 4 port USB Hub, Stereo In/Out, Ethernet 10/100...

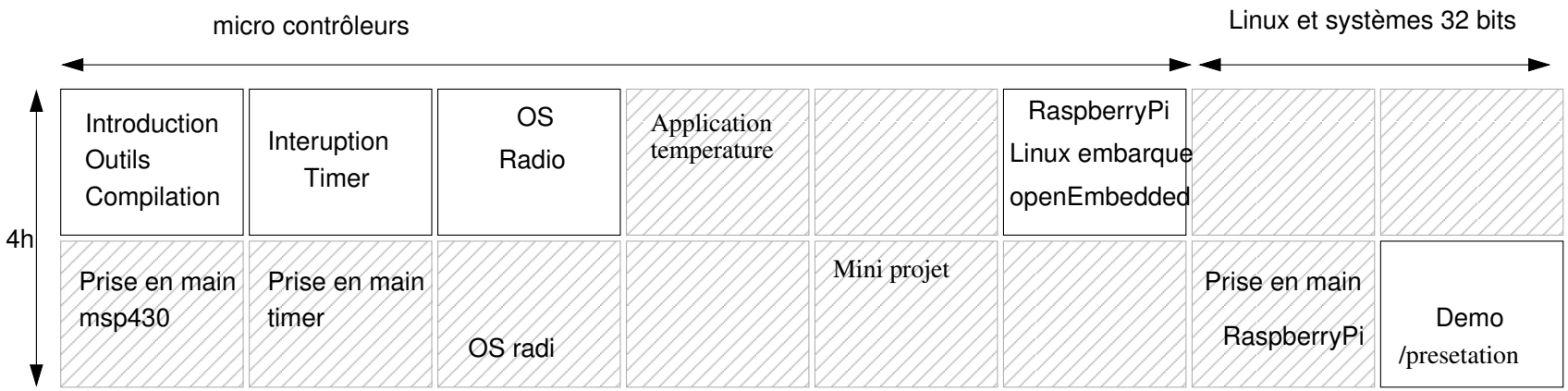
- **Rasberry Pi** ($\simeq$ 25€)
- Broadcom BCM2835, 700 MHz ARM avec FPU
- GPU Videocore 4
- RAM 512 Mo.
- 4GB SD-Card
- video RCA 2 port USB Hub, Stereo Out, Ethernet 10/100...

BED programme



- Petites objects communicants, Internet des objets
  - Capteurs, actionneurs, interface avec le monde “réél”
- Systèmes d’interconnexion, de traitements et passerelles
  - Système de localisation, de mesure
  - Transformation des données en informations
- Évaluation par groupe : présentation devant le groupe d’un miniprojet utilisant capteurs et passerelles.

BED planning



Intervention d'industriel des système embarqué.

- En 2019 :
  - Alexis Duque (Rtone, Lyon : https ://rtone.fr/)
  - Loïc Chacornac (Thales, Valence)

Évaluation : présentation des réalisations sur les plateformes

Introduction aux petits systèmes embarqués

Système informatique minimal

- Un système informatique minimal est constitué de :
  - un processeur ;
  - une mémoire ;
  - un bus de communications ;
  - des entrées/sorties (série, disque, analogique-numérique).

- Un tel système est utilisé à des fins de contrôle ou de traitement de données dans les systèmes embarqués :
  - Tâches de contrôle $\Rightarrow$ processeurs généraliste : *“micro contrôleur”*
  - Traitement de données $\Rightarrow$ DSP ou ASIP.

Embedded processors

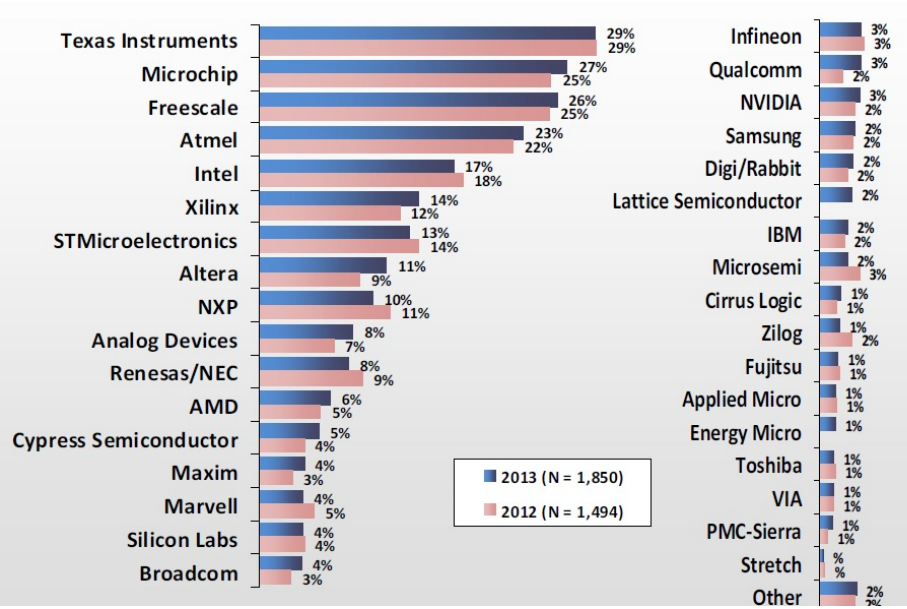
- Le terme micro-contrôleur correspond à l'utilisation du processeur, pas à sa complexité
  - Aujourd'hui un ARM7 embarqué est un micro-contrôleur
  - Il existe des micro-contrôleur 4 bits, 8 bits, 16 bits, 32 bits...
  - Les processeurs 8 bits représentent près de la moitié des processeurs vendus aujourd'hui.
- Processeurs embarqués 32 bits en 2013 (source UBM's 2013 Embedded Market Study) :

Leading MPU Suppliers ($M)

2012 Rank	Company	2011	2012	Percent Change	Percent Marketshare	Main Product Lines
1	Intel	37,435	36,892	-1%	65.3%	x86 PC, server MPUs
2	Qualcomm	4,152	5,322	28%	9.4%	ARM mobile app processors
3	Samsung (+Apple)*	2,614	4,664	78%	8.2%	ARM mobile app processors
4	AMD	4,552	3,605	-21%	6.4%	x86 PC, server MPUs
5	Freescale	1,210	1,070	-12%	1.9%	ARM and embedded MPUs
6	Nvidia	591	764	29%	1.4%	ARM mobile app processors
7	TI	510	565	11%	1.0%	ARM mobile app processors
8	ST-Ericsson**	660	540	-18%	1.0%	ARM mobile app processors
9	Broadcom	295	345	17%	0.6%	ARM mobile app processors
10	MediaTek	280	325	16%	0.6%	ARM mobile app processors

*Includes Apple's custom processors made by Samsung's foundry business.
**Cellphone IC joint venture to be dissovled by STMicroelectronics and Ericsson by 3Q13.

Source: IC Insights



Le jeu d'instructions

- Le *jeu d'instructions* (Instruction Set Architecture : ISA) a une importance capitale :
  - Il détermine les instructions élémentaires exécutées par le CPU.
  - C'est un équilibre entre la complexité matérielle du CPU et la facilité d'exprimer les actions requises
  - On le représente de manière symbolique (ex : MSP, code sur 16 bits) :

```
          mov r5,@r8 ; commentaire [R8]<-R5
lab:      ADD r4,r5  ; R5<-R5+R4
```

- Deux classes de jeux d'instructions :
  - CISC : Complex Instruction Set Computer
  - RISC : Reduced Instruction Set Computer

RISC : Reduced Instruction Set Computer

- Petites instructions simples, toutes de même taille, ayant toutes (presque) le même temps d'exécution.
- Pas d'instruction complexe.

- Accélération en pipelinant l'exécution (entre 3 et 7 étages de pipeline pour une instruction) $\Rightarrow$ augmentation de la vitesse d'horloge.
- Code plus simple à générer, mais moins compact.

- Tous les microprocesseurs modernes utilisent ce paradigme : SPARC, MIPS, ARM, PowerPC, etc.

Exemple : instructions de l'ISA du MSP

1 operand instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	opcode			B/W	Ad		Dest reg.			

relative Jumps

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	condition			PC offset (10 bits)									

2 operands instruction

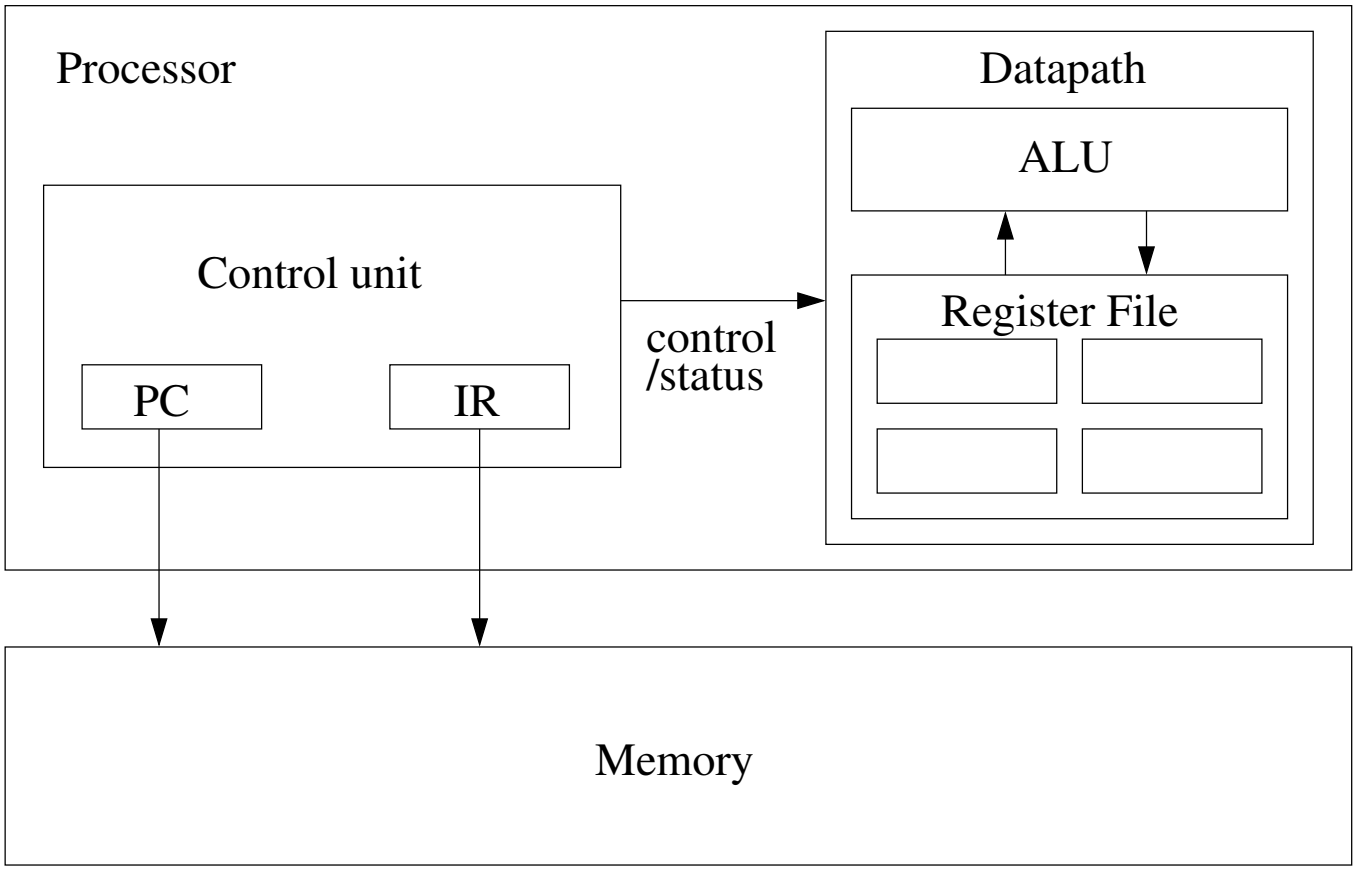
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode			Dest reg.				Ad	B/W	As			Src reg.			

Exemples :

- `PUSB.B R4`
- `JNE -56`
- `ADD.W R4,R4`

Le CPU RISC

- L'unité de contrôle configure le chemin de donnée suivant l'instruction à exécuter.
- L'exécution d'une instruction est décomposée en plusieurs phases d'un cycle : Fetch, Decode, Mem, Execute, Writeback

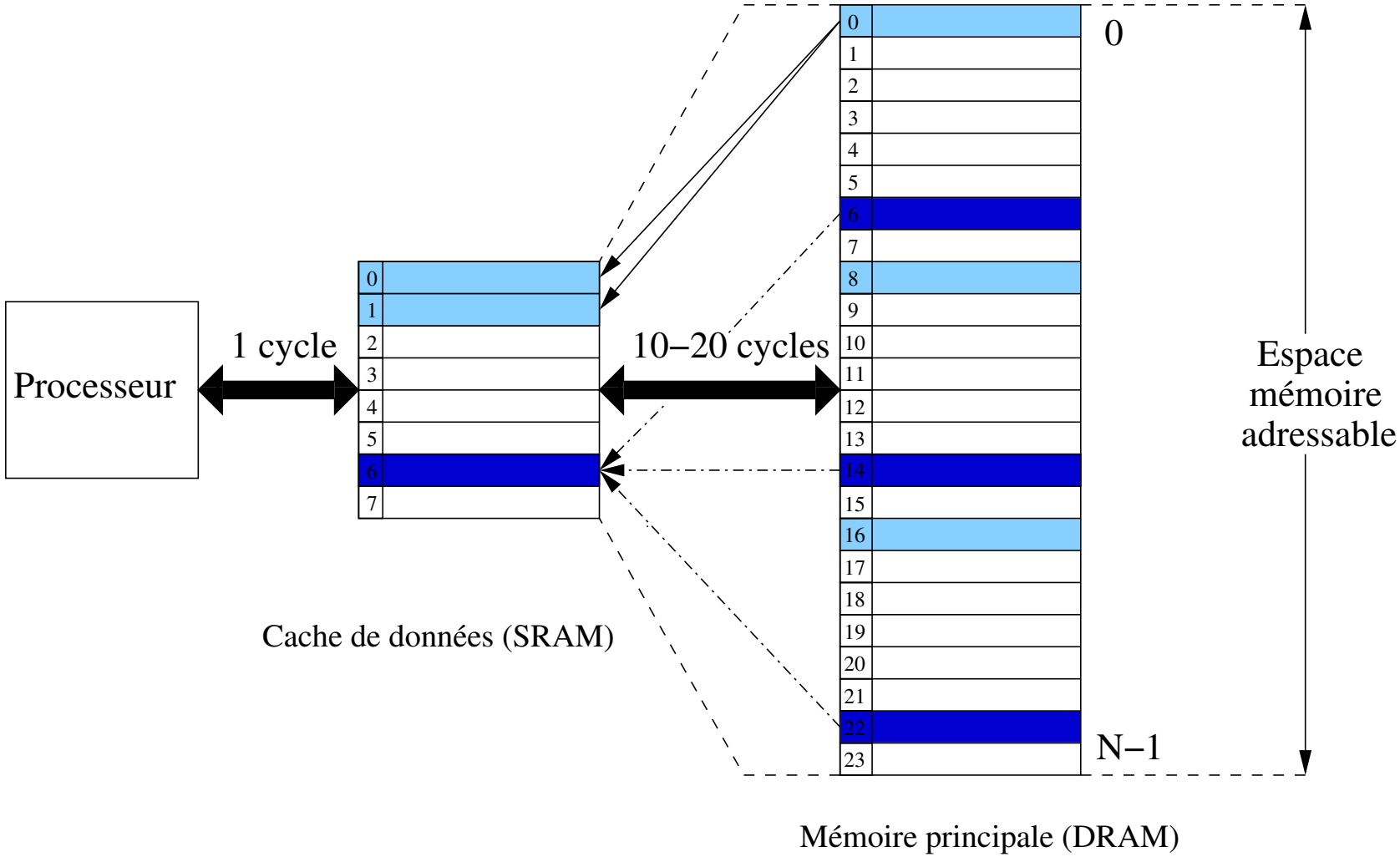


Mémoires

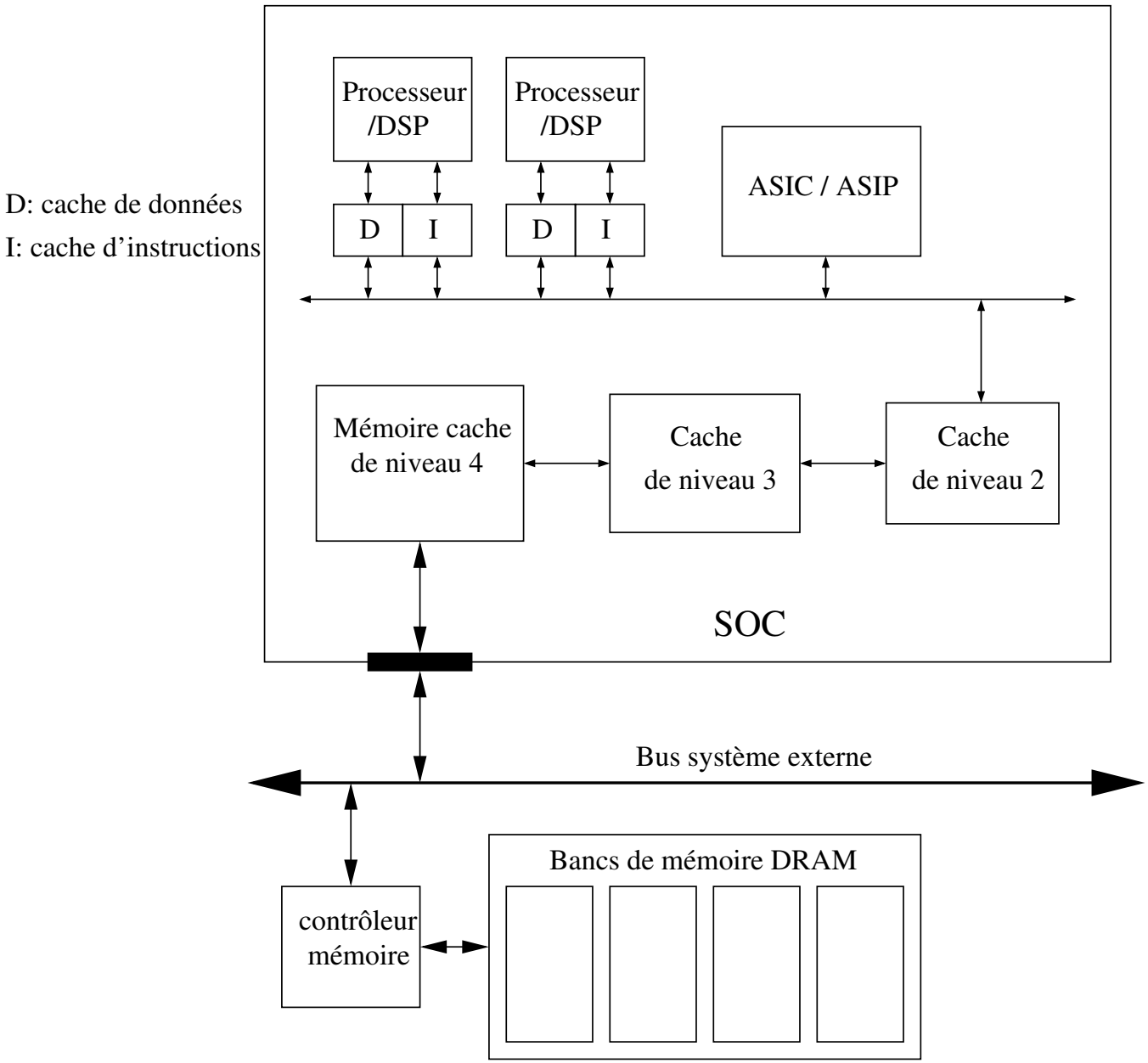
- Plusieurs technologies pour les mémoires :
  - Mémoires statiques (SRAM) : petites, rapides, consommatrices, peu denses et chères.
  - Mémoires dynamiques (DRAM) : grandes, lentes, très denses, transactions chères.

- De plus en plus de place *On-Chip* pour la mémoire (dans ce cas elles sont moins efficaces que les chips mémoire).
- Ne pas oublier que le code aussi réside en mémoire
- Tous les systèmes ont des caches pour masquer les temps de latence lors des accès à la mémoire, en général plusieurs niveaux de caches : hiérarchie mémoire.

Principe des Caches



Hiérarchie mémoire sur les SoC



Les périphériques

- Les périphériques sont accessibles :
  - Soit par des instructions spéciales ;
  - Soit (le cas le plus fréquent) par l'intermédiaire de registres situés à des adresses spécifiques, on parle de *mapping mémoire*.

- Exemple : le multiplieur matériel du msp430
  - Accessible par des registres mappés entre les adresses `0x0130` et `0x013F`
  - Écriture à l'adresse `0x130`, positionne le premier opérande
  - Écriture à l'adresse `0x138`, positionne le deuxième opérande et lance le calcul
  - Le résultat est à l'adresse `0x013A`, sur 32 bits

Mapping mémoire : exemple du multiplieur câblé

```
int main(void) {
    int i;
    int *p,*res;

    p=0x130;
    *p=2;
    p=0x138;
    *p=5;
    res=0x13A;
    i=*res;

    nop();
}
```

```
int main(void) {
    int i;
    int *p,*res;

    __asm__("mov #304, R4");
    __asm__("mov #2, @R4");
    // p=0x130;
    // *p=2;
    __asm__("mov #312, R4");
    __asm__("mov #5, @R4");
    // p=0x138;
    // *p=5;
    __asm__("mov #314, R4");
    __asm__("mov @R4, R5");
    // res=0x13A;
    i=*res;

    nop();
}
```

Entrées/Sorties

- Certains micro-contrôleur possèdent des périphériques intégrés permettant des facilités de communications (ex : UART, SPI...)
- En général, on a un accès brut aux broches du circuit
- Exemple : extrait de `msp430x22x4.h`

```
[...]
#define P1IN              0x0020    /* Port 1 Input                    */
#define P1OUT             0x0021    /* Port 1 Output                   */
#define P1DIR             0x0022    /* Port 1 Direction                */
#define P1IFG             0x0023    /* Port 1 Interrupt Flag           */
#define P1IES             0x0024    /* Port 1 Interrupt Edge Select */
#define P1IE              0x0025    /* Port 1 Interrupt Enable         */
#define P1SEL             0x0026    /* Port 1 Selection                */
#define P1REN             0x0027    /* Port 1 Resistor Enable          */
[...]
```

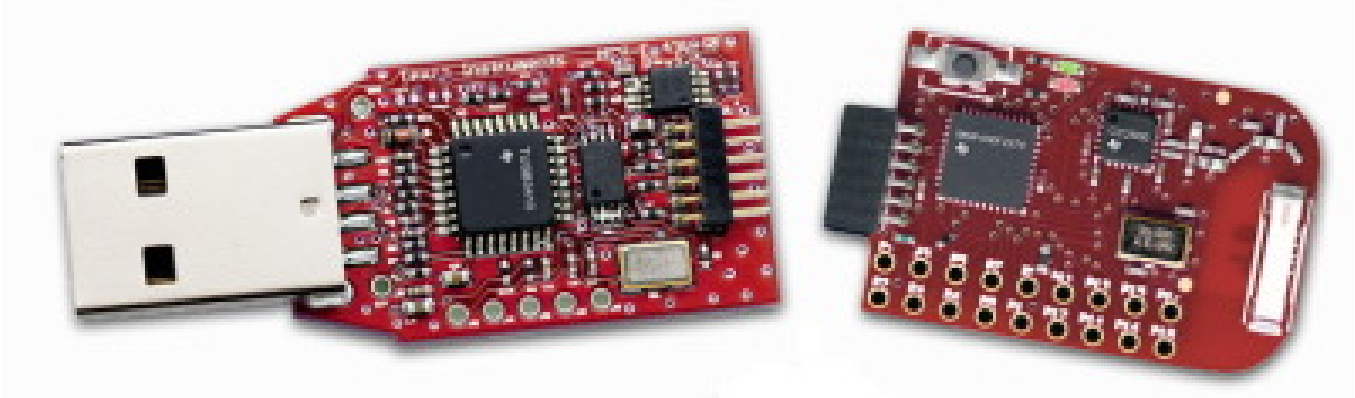
Low Power Mode pour le MSP430

- Différent mode pour réduire la consommation
  - LPM0 : le CPU est arrêté
  - LPM1, LPM2 : l'horloge rapide (`SMCLK`) est aussi arrêtée
  - LPM3 : le générateur d'horloge est arrêté
  - LPM4 : l'oscillateur du cristal est arrêté

- Le temps de reprise est d'autant plus long que la veille est profonde.

Introduction au MSP430

Présentation de la clé ez430-rf2500

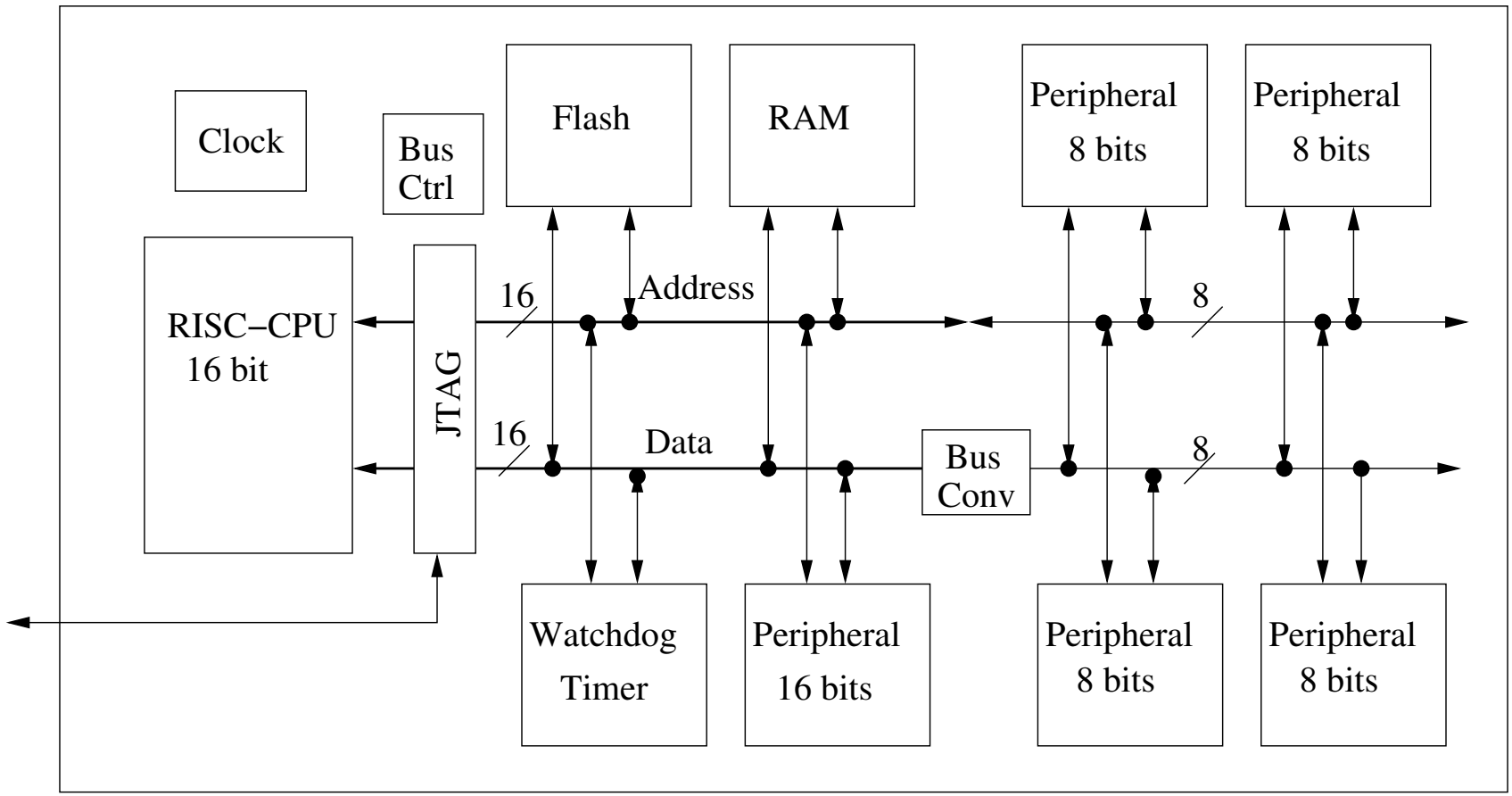


- Le module eZ430-RF2500 est équipé d'un MSP430f2274 et d'un chip radio CC2500 (transceiver low power de chipcon destiné à la bande ISM 2.4-GHz)
- Le module eZ430-RF2500 est vendu avec un kit de développement consistant en une clé usb permettant
  - de programmer le msp430
  - de débugger en direct avec gdb
  - de récuperer la sortie du port série du MSP430

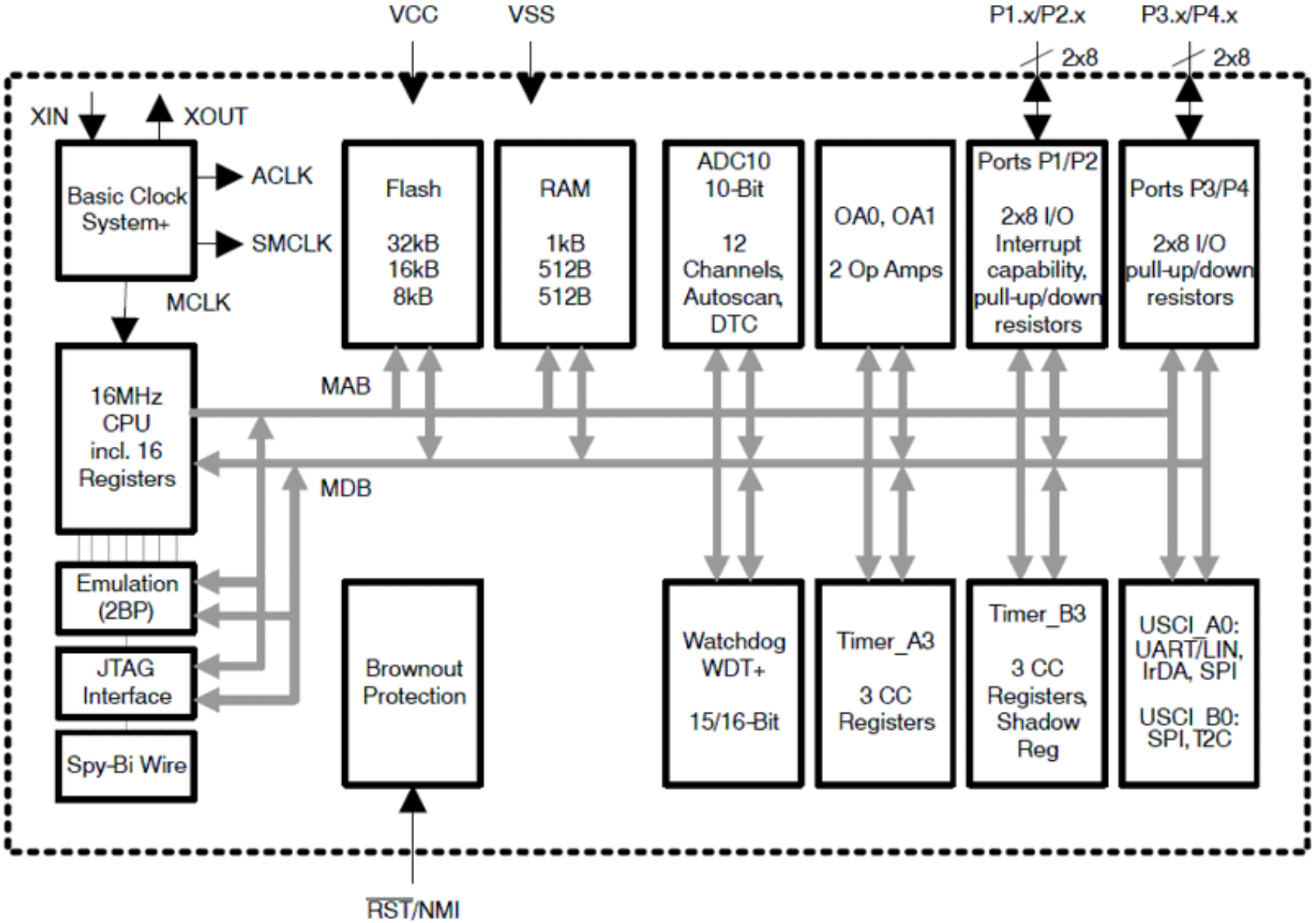
Documents techniques

- Documents du cours
- Manuel de programmation du MSP
  `MSP430x2xx_Family_User's_Guide_(Rev._D)_slau144d.pdf`
- Manuel du MSP430F2274 `msp430f2274.pdf`
- Feuilles de schématique
- User's Manual EZ430
  `eZ430-RF2500_UserGuide_SLAU227A.pdf`
- Datasheet du composant radio `cc2500.pdf`
- Site de Texas Instrument
  - Exemples de programme / drivers, Notes d'application, ...
- The `mspgcc` toolchain : `http://mspgcc.sourceforge.net/`
- Notamment la doc `mspgcc`
  (`http://mspgcc.sourceforge.net/manual/`)

Architecture du MSP



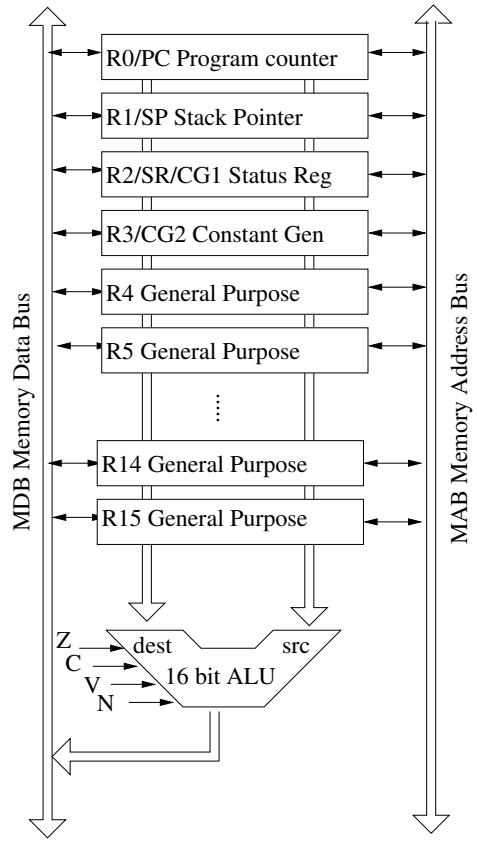
Architecture du MSP



Note: Memory, peripherals, and ports may vary depending on the device.

CPU RISC 16 bits

- 28 Instructions sur 16 bits
- 64 Ko de mémoire adressable
- Périphériques mappés en mémoire
- 16 registres 16 bits (r0-r16)
  - r0 : PC (Program counter)
  - r1 : SP (Stack pointeur)
  - r2 : SR (status register)
  - r3 : constante 0



R3: Status Register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved							V	SCG0	SCG1	OSC OFF	CPU OFF	GIE	N	Z	GC

Périphérique mappés en mémoire

- l'écriture à une certaine adresse est interprétée comme une communication avec le périphérique.
- Exemple : le multiplieur matériel accessible par des registres mappés entre les adresses `0x0130` et `0x013F`
- Les autres périphériques sont aussi accessibles par des registres mappé en mémoire : les SFR (Special Function Registers), en C :
  - écriture vers le périphérique : `SFR = valeur`
  - lecture des registres du périphérique : `variable = SFR`

Extrait de `/usr/msp430/include/msp430x22x4.h` :

```
#define sfrw_(x,x_) volatile unsigned int x __asm__("__" #x)
#define sfrw(x,x_) extern sfrw_(x,x_)
[...]
#define TACTL_                0x0160    /* Timer A Control */
sfrw(TACTL, TACTL_);
[...]
```

équivalent à

```
extern volatile unsigned int TACTL __asm__("__0x0160");
```

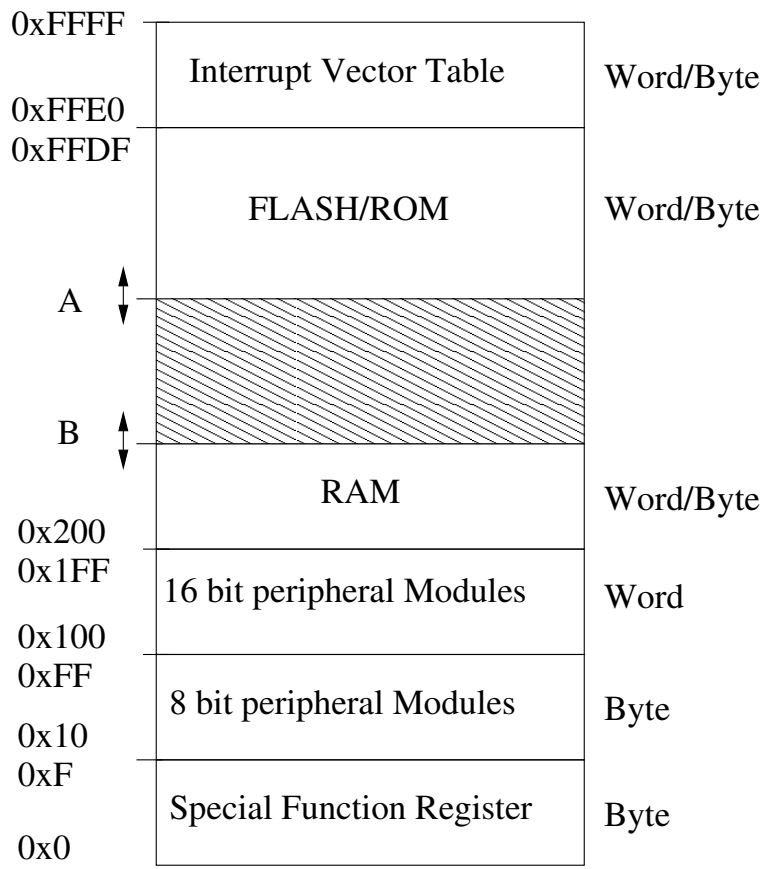
équivalent à

```
#define TACTL (*(int *)0x0160)
```

Mapping mémoire MSP439F2274

Sur le MSP430F2274 :

- `0x0000` à `0x01FF` : périphériques
- `0x0200` à B=`0x05FF` : RAM (1Ko), Données et pile d'exécution
- `0x0C00` à `0x0FFF` : Boot mem (1Ko, ROM).
- `0x1000` à `0x10FF` : byte info. mem. (256 octets, Flash)
- `0x8000` à `0xFFFF` : Code (32 Ko, Flash).
  - Dont `0xFFE0` à `0xFFFF` vecteurs d'interruption



Principe du mécanisme d'interruption

- Par défaut, le programme `main` est exécuté à l'infini, il contient en général une boucle infinie pour ne jamais terminer.
- Le processeur peut recevoir à tout instant des *interruptions* (sous-entendu *interruptions matérielles*).
- Une interruption peut être envoyée par un périphérique du MSP (timer, chip radio, port série, etc...), ou reçue de l'exterieur (sur un GPIO) comme le `reset` par exemple.
- C'est le programmeur qui configure les périphériques (par exemple le timer) pour envoyer un interruption lors de certains évennements
- Qu'elles soient internes ou externes les interruptions arrivent sur un port du MSP.
- Elle est traitée par un *handler d'interruption* (ou *interrupt service routine* : ISR).
- Chaque interruption possède sa propre ISR. c'est une fonction écrite par le programmeur qui a des propriété un peu particulières.

Handler d'interruption (ISR)

- Le programme (qui a configuré les périphériques pour envoyer des interruption) configure les ports correspondant du MSP pour *autoriser* les interruptions (sinon l'interruption sera ignorée).
- Il écrit le handler d'interruption (ISR) : fonction C qui sera exécutée lors de l'interruption. Elle a les caractéristiques suivantes :
  - Elle n'a pas de paramêtre.
  - Elle doit être compilée de manière un peu différente, elle est donc en général identifiée par un *pragma* à destination du compilateur. Example pour `gcc` :
    ```
    interrupt(PORT1_VECTOR)port1_irq_handler(void)
    ```
    Elle peut aussi être écrite directement en assembleur ;
  - Elle doit être courte (en général ininterruptible)
- l'appel à une fonction d'interruption est (autant que possible) transparente pour l'exécution du programme. En général elle modifie une variable d'état du programme (par exemple : un drapeau indiquant un paquet reçu ou l'état des leds). Lorsqu'elle se termine le programme reprend son exécution là ou il a été interrompu

Vecteurs d'interruption (fin de la mémoire)

- Pour connaître l'adresse ou est stocké le code d'un ISR, le processeur se réfère à *latable de vecteur d'interruption* (Interrupt Vector Table, cf mapping mémoire, p. 29)
- *Interrupt Vector Table* : indique les adresses des fonctions de gestion des interruptions.
- Par exemple, l'adresse du vecteur d'interruption du reset est `0xFFFE`.
- Lors du Boot, le MSP va lire à l'adresse `0xFFFE`, qui contient l'adresse de l'ISR du reset.
- Lorsque l'on écrit une ISR (*Interrupt Service Routine*), par exemple pour rattraper l'interruption du timer, le compilateur met en place l'adresse de la fonction écrite dans le vecteur d'interruption correspondant.

Instructions de l'ISA du MSP30

- Jeu d'instruction RISC 16 bits, trois formats d'instructions :
  - Les instructions à un opérande (8)
  - Les sauts (8)
  - Les instructions à deux opérandes (opcode de $0100_2 = 4_{10}$ à $1111_b = 16_{10}$) (12)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	Opcode			B/W	Ad		Dest reg			
0	0	1	Condition			PC offset (10 bit)									
Opcode				Source reg				Ad	B/W	As		Dest reg			

- Soit au total 28 instructions
- (presque) chaque instruction `inst` peut être écrite `inst.b` (version octet) ou `inst.w` (version mot)

Instructions à un opérande

000	RRC(.B)	9-bit rotate right through carry. C$\rightarrow$ msbit $\rightarrow$...$\rightarrow$ lsbit $\rightarrow$C. Clear the carry bit beforehand to do a logical right shift.
001	SWPB	Swap 8-bit register halves. No byte form.
010	RRA(.B)	Badly named, this is an 8-bit arithmetic right shift.
011	SXT	Sign extend 8 bits to 16. No byte form.
100	PUSH(.B)	Push operand on stack. Push byte decrements SP by 2.
101	CALL	Fetch operand, push PC, then assign operand value to PC. Note the immediate form is the most commonly used. This has no byte form.
110	RETI	Pop SP, then pop PC. The operand field is unused.
111	Not used	The MSP430 actually only has 27 instructions.

Sauts relatifs

000	JNE/JNZ	Jump if Z==0 (if !=)
001	JEQ/Z	Jump if Z==1 (if ==)
010	JNC/JLO	Jump if C==0 (if unsigned <)
011	JC/JHS	Jump if C==1 (if unsigned >=)
100	JN	Jump if N==1 Note there is no "JP" if N==0 !
101	JGE	Jump if N==V (if signed >=)
110	JL	Jump if N !=V (if signed <)
111	JMP	Jump unconditionally

Instructions à deux opérandes

0100	MOV src,dest	dest = src	The status flags are NOT set.
0101	ADD src,dest	dest += src	
0110	ADDC src,dest	dest += src + C	
0111	SUBC src,dest	dest +=˜src + C	
1001	SUB src,dest	dest -= src	Implemented as dest +=˜src + 1.
1001	CMP src,dest	dest - src	Sets status only ; the destination is not written.
1010	DADD src,dest	dest += src + C, BCD.	Decimal add
1011	BIT src,dest	dest & src	Sets status only ; the destination is not written.
1100	BIC src,dest	dest &=˜src	The status flags are NOT set.
1101	BIS src,dest	dest |= src	The status flags are NOT set.
1110	XOR src,dest	dest ˆ= src	
1111	AND src,dest	dest &=- src	

Instructions émulées

NOP	MOV r3,r3
POP dst	MOV @SP+,dst
BR dst	MOV dst,PC
RET	MOV @SP+,PC
CLRC	BIC #1,SR
SETC	BIS #1,SR
CLRZ	BIC #2,SR
SETZ	BIS #2,SR
CLRN	BIC #4,SR
SETN	BIS #4,SR
DINT	BIC #8,SR
EINT	BIC #8,SR

RLA(.B) dst	ADD(.B) dst,dst
RLC(.B) dst	ADDC(.B) dst,dst
INV(.B) dst	XOR(.B) #-1,dst
CLR(.B) dst	MOV(.B) #0,dst
TST(.B) dst	CMP(.B) #0,dst
DEC(.B) dst	SUB(.B) #1,dst
DECD(.B) dst	SUB(.B) #2,dst
INC(.B) dst	ADD(.B) #1,dst
INCD(.B) dst	ADD(.B) #2,dst
ADC(.B) dst	ADDC(.B) #0,dst
DADC(.B) dst	DADD(.B) #0,dst
SBC(.B) dst	SUBC(.B) #0,dst

Modes d'adressage

- L'opérande source (ou le seul opérande pour les instructions à un opérande) est codé avec 2 bits pour le mode d'adressage et 4 bits pour le numéro de registre :

codage	syntaxe	appellation
00 nnnn	Rn	Registre direct
01 nnnn	offset(Rn)	Registre indéxé
10 nnnn	@Rn	Registre indirect
11 nnnn	@Rn+	Registre indirect avec post-incrément

- L'opérande destination n'a qu'un bit pour le mode d'adressage qui peut être soit direct, soit indéxé (et donc indirect avec un offset de 0)
- Exemple, l'instruction C :
  `*p++ *= 2;` $\Leftrightarrow$ `add @Rn+,-2(Rn)`

Adressage de R2 et R3

- Lorsque R2 (status register) or R3 (zero register) sont adressés, le mode d'addressage est décodé de manière particulière :

code	opérande	sémantique
00 0010	r2	Normal access
01 0010	&<location>	Absolute addressing. the usual PC-relative addressing should not be used.
10 0010	#4	This encoding specifies the immediate constant 4.
11 0010	#8	This encoding specifies the immediate constant 8.
00 0011	#0	This encoding specifies the immediate constant 0.
01 0011	#1	This encoding specifies the immediate constant 1.
10 0011	#2	This encoding specifies the immediate constant 2.
11 0011	#-1	This specifies the all-bits-set constant, -1.

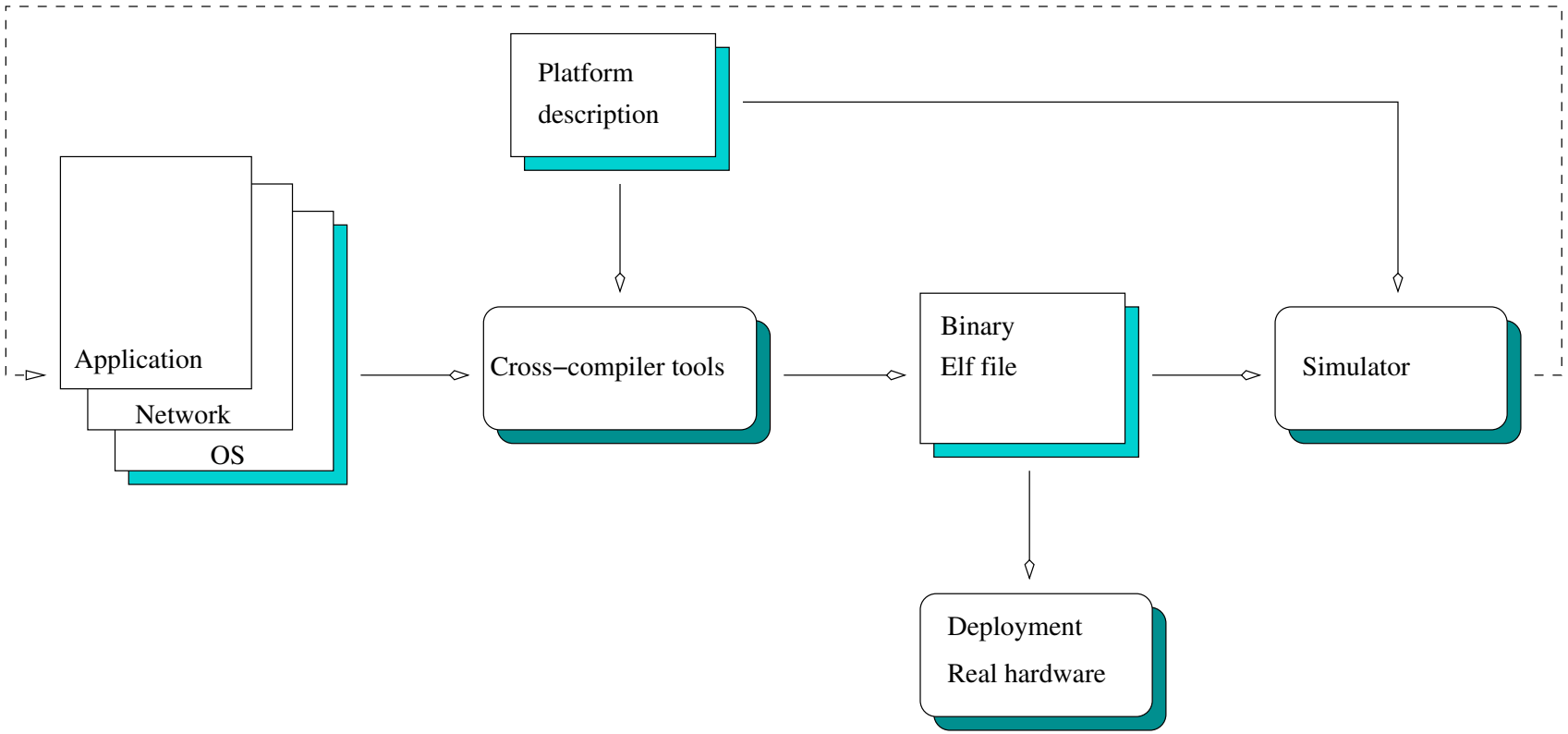
Durée des instructions

- La règle générale est que une instruction prend un cycle par mot de mémoire accédé.
- Donc, un cycle pour l'instruction, un cycle pour une source en mémoire, un cycle pour une destination en mémoire et un cycle par mot d'offset
- Exceptions :

PUSH Rn	3 cycles
PUSH @Rn, @Rn+, #x	4 cycles
PUSH offset(Rn)	5 cycles
CALL Rn	4 cycles
CALL @Rn	4 cycles
CALL @Rn+, #x	5 cycles
CALL offset(Rn)	5 cycles
RETI	5 cycles
Interruption	6 cycles
Reset	4 cycles

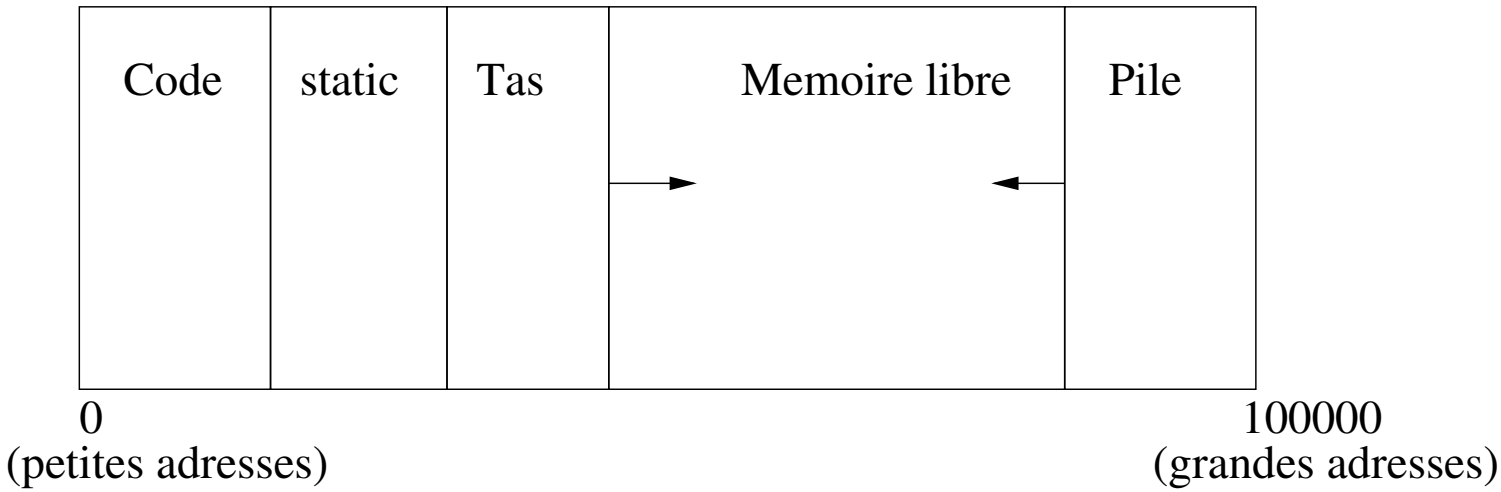
Compilation pour l'embarqué

Compilation



Pile d'exécution

- Le mécanisme de transfert de contrôle entre les procédures est implémenté grâce à la *pile d'exécution*.
- Le programmeur à cette vision de la mémoire virtuelle :

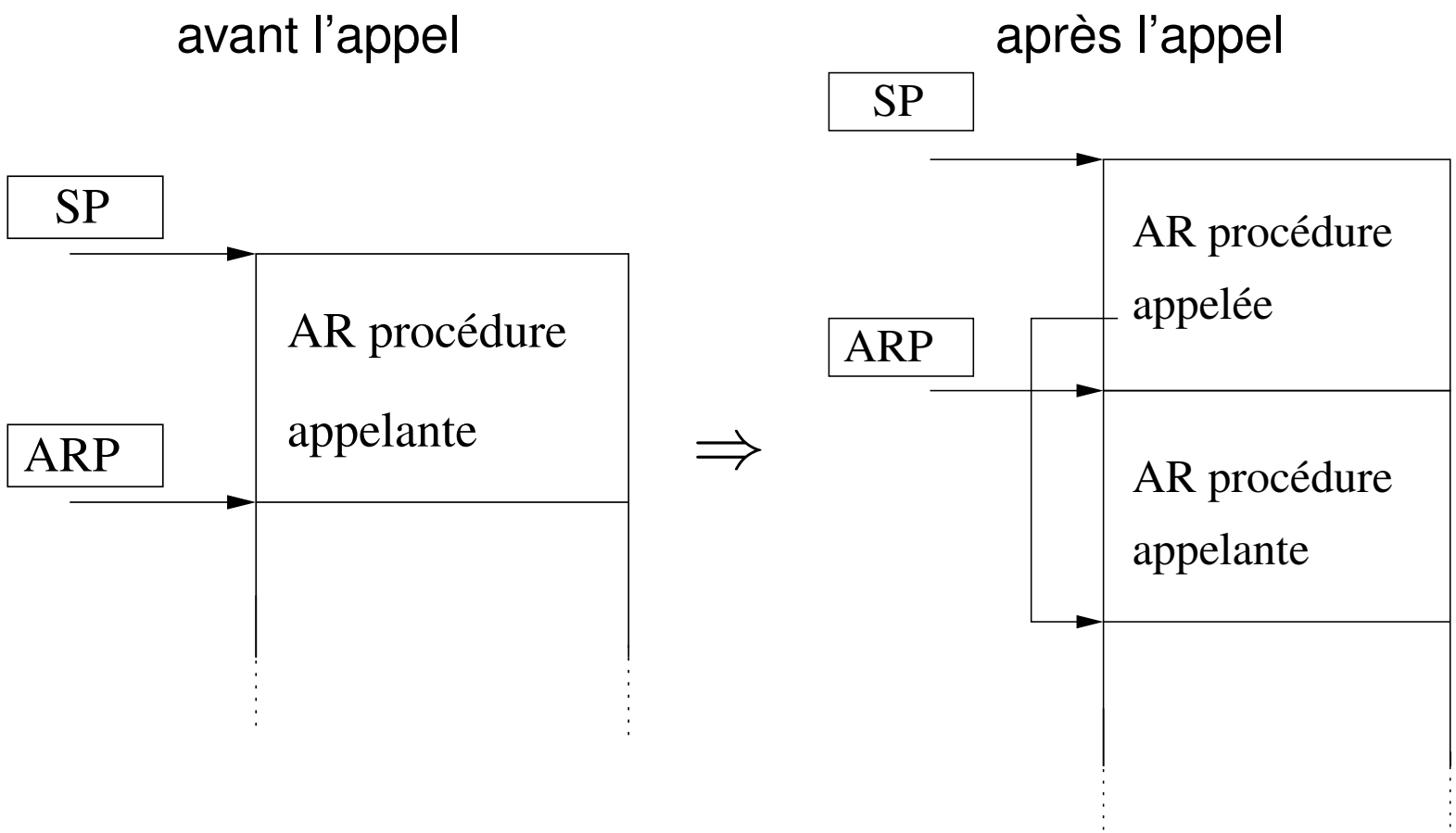


- Le tas (*heap*) est utilisé pour l'allocation dynamique.
- La pile (*stack*) est utilisée pour la gestion des contextes des procédures (variable locales, etc.)

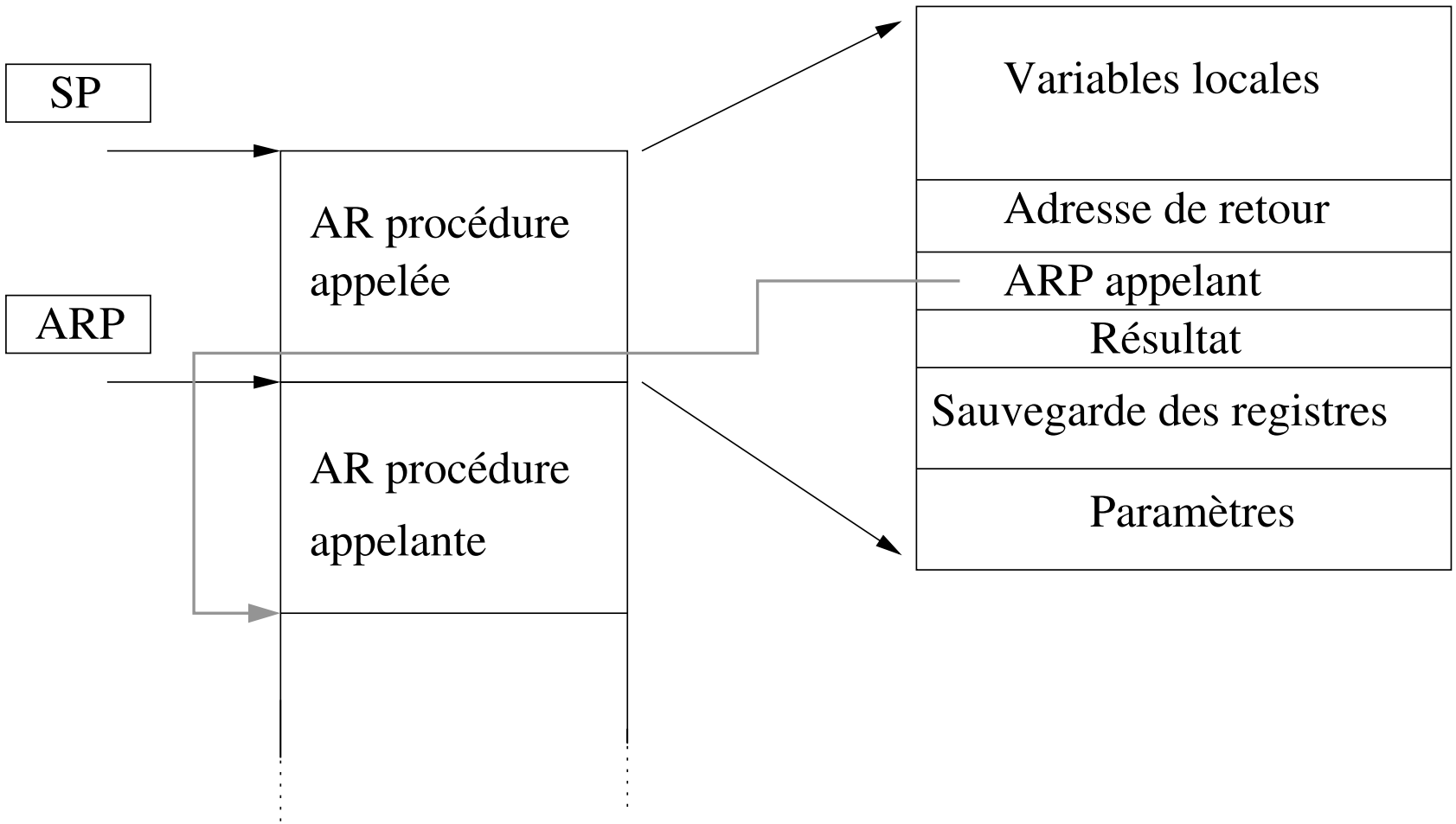
Enregistrement d'activation

- Appel d'une procédure : empilement de l'*enregistrement d'activation* (AR pour *activation record*).
- L'AR permet de mettre en place le *contexte* de la procédure.
- Cet AR contient
  - L'espace pour les variables locales déclarées dans la procédure
  - Des informations pour la restauration du contexte de la procédure appelante :
    - Pointeur sur l'AR de la procédure appelante (ARP ou FP pour *frame pointeur*).
    - Adresse de l'instruction de retour (instruction suivant l'appel de la procédure appelante).
    - Éventuellement sauvegarde de l'état des registres au moment de l'appel.

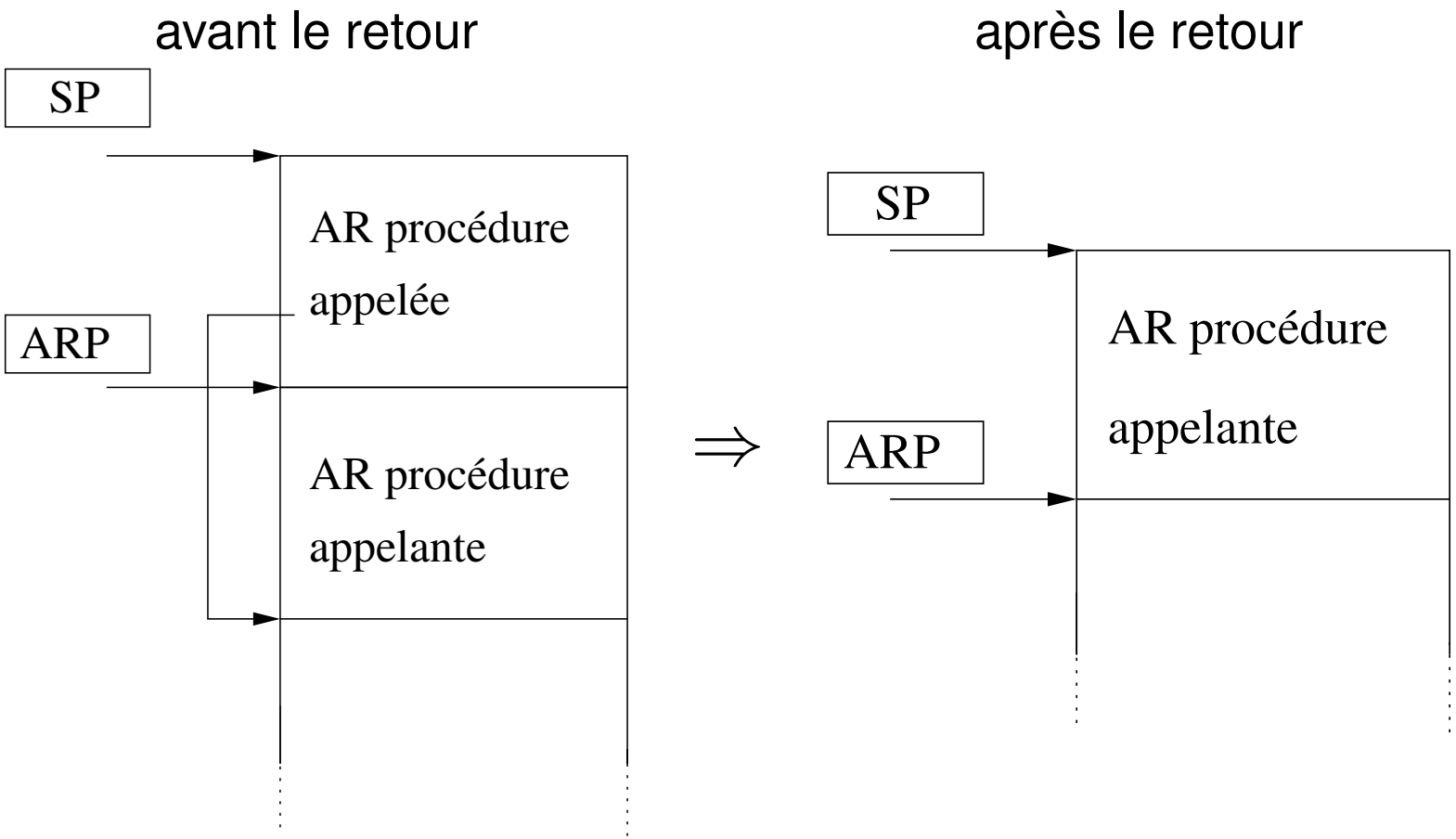
Appel de procédure : état de la pile



Contenu de l'AR



Retour de procédure : état de la pile



Convention d'appel

- Le mécanisme de Pile n'est pas *normalisé*, son implantation précise est déterminée par le compilateur,

- Pour pouvoir interfacer des fonctions compilées avec un certain compilateur (par exemple, les utiliser en librairie), il faut utiliser les mêmes *conventions d'appel* (*calling convention* ou *ABI : Application Binary Interface*).

- La convention d'appel est un accord entre l'OS, le compilateur et l'ISA. En général elle est spécifiée par le compilateur.

Convention d'appel MSP430 de GCC

- Pour le compilateur GCC les conventions d'appel sont les suivantes :
  - R0 est le compteur de programme (*program counter : PC*)
  - R1 est le pointeur de pile (*Stack pointer SP*)
  - R4 est l'ARP (*Frame pointer FP*
  - Les quatre premiers arguments d'une fonction sont passés par les registres R12, R13, R14 et R15.
  - Ces quatre registres (R12, R13, R14 et R15) sont *clobbered* (ou *callee save*) : il ne sont pas sauvegardés lors d'un appel de fonction.
  - Les registres R6-R11 sont *caller save* il sont sauvegardés sur la pile (si besoin) lors d'un appel de fonction
  - R5 est le pointeur d'argument : pointe sur le premier argument passé sur la pile
  - R15 est utilisé pour transmettre le résultat (R14 :R15 dans le cas d'un type 32 bits)

Convention d'appel MSP430 de IAR

- R0 est le compteur de programme (*program counter : PC*)
- R1 est le pointeur de pile (*Stack pointer SP*)
- Il n'y a pas de d'ARP (directive CFI pour *call frame Information*).
- Les deux premiers arguments d'une fonction sont passés par les registre R12, R14 (+R13 et R15 pour 32 bits) ils ne sont pas sauvegardés.
- Les registres R4-R11 sont sauvegardés sauf si R8-R11 sont utilisés pour passer des paramètres 64 bits
- R15 est utilisé pour transmettre le résultat (R14 :R15 dans le cas d'un type 32 bits)
- Lors d'un appel, la fonction appelante empile dans l'enregistrement d'activation :
  - Les paramètres (sauf les deux premiers).
  - l'adresse de retour.
  - Les registres sauvegardés.

Interruptions

- Les interruptions (sous-entendu “matérielles”) sont indispensables au fonctionnement de tout ordinateur.

- Lorsqu’une interruption survient, le microprocesseur sauvegarde l’état courant de son programme en cours d’exécution :
  - tous les registre généraux
  - le registre d’état
  - le program counter

- Il exécute ensuite une portion de code spécifique pour traiter cette interruption (interrupt handler)
- lorsque le handler est terminé, il restaure l’état du processeur et reprend l’exécution du programme interrompu

Interruptions (2)

- L'appel à la routine de traitement de l'interruption n'est pas exactement un appel de fonction comme les autres.
- L'implantation exacte du processus de *rattrapage* des interruptions fait partie de la convention d'appel.
  - Par exemple, Sur le MSP, le handler d'interruption termine par l'instruction RETI (return from interrupt).
- un handler d'interruption peut, lui même, être interrompu ou pas par une autre interruption (priorité des interruptions).
- L'utilisateur peut écrire ses propre routines d'interruption en C, les compilateurs fournissent des facilités pour cela.

Gestionnaire d'interruption

- Les compilateur ne peuvent généralement pas directement compiler un gestionnaire d'interruption car ils suivent une convention d'appel spéciale. Pour le MSP :
  - Retour par `reti`
  - Il faut sauvegarder tous les registres sur la pile
  - Il faut ignorer les bit C/N/Z/O de SR
- Pour écrire un handler d'interruption, on peut :
  - Encapsuler du code C dans un *wrapper* spécial
  - Écrire la routine d'interruption en assembleur
  - (Le plus fréquent) Utiliser des fonctions intrinsèques du compilateur :

    Pour GCC :

    ```
    interrupt(PORT1_VECTOR)port1_irq_handler(void)
    ```

    Pour IAR :

    ```
    #pragmavector=PORT1_VECTOR
    __interruptvoidport1_irq_handler(void)
    ```

ISR pour MSP430 avec gcc

```
interrupt (PORT1_VECTOR) port1_irq_handler(void)
{
  if (P1IFG & (P1IE & (1 << 2)))
    {
      SWITCH_RED_LED();
    }
  P1IFG = 0;
}
```

ISR pour MSP430 avec IAR

```
#pragma vector=TIMERA1_VECTOR
__interrupt void Timer_A1_ISR(void)
{
  switch (__even_in range(TAIV, 10))
     {
     case 2: P1POUT ^= 0x04;
             break;
     case 4: P1POUT ^= 0x02;
             break;
     case 10: P1POUT ^= 0x01;
     break
    }
}
```

Mémoires des systèmes embarqués

- Il existe plusieurs types de mémoires qui n'ont pas les mêmes caractéristiques (rémanente ou pas, latence d'accès, densité, consommation, etc.).
- Il est important de contrôler l'emplacement physique ou seront stocké : le code, certaines données etc.
- Un code exécutable est organisé en *sections* ou *segments*.
- L'éditeur de lien rassemble différents *morceaux* de segment et les *place* à une certaine adresse.
- Ce processus est guidé par un fichier de commande (*linker script*) paramétré par l'utilisateur : :
  - Pour `GCC` (`ld`) c'est le fichier `ld_script`.
  - Pour IAR c'est le fichier `lnk430F2274.xcl`

Section pour GCC

- Par défaut, GCC utilise les sections suivantes :
  - `.text` : qui contient les instructions et qui sera stockée en flash
  - `.data` : qui contient les variables initialisées dans le programme (exemple : “`int i=2;`”) et qui sera stockée en flash mais sera transférée dans la RAM à l’exécution du programme
  - `.bss` : qui contient les variables non initialisées (exemple : "int i ;") et qui sera stockée dans la RAM.
- De nombreuses autres sections sont créées notamment pour les informations de debug.

Exemple de LD script pour GCC (MIPS)

- Spécification de fichier de librairie
- Spécification du format de sortie
- 5 sections nommées

```
/* Une list de fichier à inclure (les autres sont
     spécifiés par la ligne de commande */
INPUT(libc.a libg.a libgcc.a libc.a libgcc.a)

/* Specification du format de sortie
   (binaire:bin}, Intel Hex:ihex, debug coff-\$target */
OUTPUT_FORMAT("coff-sh")

/* list of our memory sections */
MEMORY {
 vect : ORIGIN = 0x00000000, LENGTH = 1k
 rom  : ORIGIN = 0x00000400, LENGTH = 127k
 reset: ORIGIN = 0xBFC00000, LENGTH = 0x00000400
 ram  : ORIGIN = 0x400000,   LENGTH = 128k
 cache: ORIGIN = 0xfffff000, LENGTH = 4k
}
```

Exemple de LD script (suite)

- Placement de chaque section en mémoire
- Création d'un symbole en début de section (ex : `__text_start`) et en fin (`__text_end`)
- Placement des parties du code préfixé par la directive `.text` dans la section `rom` de la mémoire.
- Éventuellement insertion de code spécifiquement écrit directement dans la mémoire (`.reset`)

```
SECTIONS {
 /* the interrupt
    vector table */
 .vect :
 {
  __vect_start = .;
  *(.vect);
  __vect_end = .;
 } > vect

/* code and constants */
 .text :
 {
  __text_start = .;
  *(.text)
  *(.strings)
   __text_end = .;
 } > rom

.reset : {
  ./libhandler.a(.reset)
 } > reset
```

```
/* uninitialized data */
 .bss :

 {
   __bss_start = . ;
  *(.bss)
  *(COMMON)
   __bss_end = . ;
 } > ram

/* initialized data */
 .init : AT (__text_end)
 {
  __data_start = .;
  *(.data)
  __data_end = .;
 } > ram

/* application stack */
 .stack :
 {
   __stack_start = .;
  *(.stack)
   __stack_end = .;
 } > ram
}
```

Contrôler les sections du programme

- La répartition des données dans les sections peut être aussi contrôlée depuis le programme source.
- Le concepteur de logiciel embarqué veut souvent contrôler explicitement la répartition des variables globales dans les sections.
- Utilisation de la directive `__attibute__` pour `GCC`

```
const int put_this_in_rom
      __attribute__((section("myconst" )));
const int put_this_in_flash
      __attribute__((section("myflash" )));
```

- Utilisation de directives nommées comme les sections par `IAR` (IAR compiler référence guide p 35)

```
__data16 int j; // j to be put in segment DATA16_Z
__no_init _data16 int i; //i to be put in segment DATA16_N
__data16 int k=7// j to be put in segment DATA16_I in RAM
                // initializer in DATA16_ID in ROM
```

Où trouver des informations

- Manuel de programmation du MSP : **Moodle**

  `MSP430x2xx_Family_User's_Guide_(Rev._D)_slau144d.pdf`
- Manuel du MSP430F2274 : **Moodle**

  `msp430f2274.pdf`
- Macro MSP
  - `mspgcc` installé sur les machines du département.
  - Notamment : `/usr/msp430/include/msp430f2274.h`
- ABI et compilation avec mspgcc
  - The `mspgcc` toolchain : `http://mspgcc.sourceforge.net/`
  - Notamment la doc `mspgcc` (`http://mspgcc.sourceforge.net/manual/`)